СЕРИЯ

БИБЛИОТЕКА ПРОГРАММИСТА



Perl & XML

Jason McIntosh & Erik T. Ray



Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Perl & XML

БИБЛИОТЕКА ПРОГРАММИСТА

Джейсон Макинтош
Эрик Т. Рэй



Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков • Минск
2003

ББК 32.973-018
УДК 681.3.06
Р96

Р96 **Perl & XML. Библиотека программиста** / Э. Рэй, Дж. Макинтош. — СПб.: Питер, 2003.— 208 с: ил.

ISBN 5-94723-482-3

Книга посвящена всестороннему рассмотрению особенностей создания XML-приложений средствами языка Perl. Рассматриваются теоретические основы генерирования и синтаксического разбора XML-документов, обработка XML-деревьев, объектная модель документов (DOM), работа с потоками событий, а также Perl-модули различного назначения. Серьезный теоретический материал иллюстрируется большим количеством практических примеров. Книга рассчитана на программистов, имеющих опыт работы на языке Perl.

ББК 32.973-018
УДК 681.3.06

ISBN 0-596-00205-X (англ.)
ISBN 5-94723-482-3

Краткое содержание

Содержание

Введение

В книге рассматриваются темы, находящиеся на стыке двух важнейших технологий, применяемых в Web и для поддержки функционирования информационных служб. На сегодняшний день XML является самым современным и наиболее приемлемым языком разметки для работы с данными, содержащими описание собственной структуры. В распоряжении программиста оказываются средства, позволяющие реализовать обобщенный формат представления данных. Выбор языка написания сценариев Perl для обработки XML-кода представляется вполне естественным (особенно если учесть то, что Perl длительное время применяется web-программистами для комбинирования разнородных компонентов и генерирования динамического содержимого в Сети).

Язык XML обладает гораздо большими возможностями, чем HTML, по в то же время он менее требователен к ресурсам, чем SGML. Благодаря этому его использование целесообразно для решения многих задач. Этот язык, в силу своей гибкости, пригоден для описания информации любого типа, от обычных web-страниц до юридических контрактов на издание книг. Точности, которая достигается благодаря XML-описанию, вполне достаточно для форматирования данных средствами. предлагаемыми такими информационными службами, как SOAP и XML-RPC. Также реализована поддержка распространенных стандартов, например кодировки Unicode. Благодаря универсальности этой кодировки обеспечивается обратная совместимость с кодировкой ASCII. Несмотря на разнообразие возможностей, язык XML поразительно прост в применении. В силу этого многие разработчики рассматривают его в качестве некоего «эликсира молодости», способного придать новые качества старым программам.

Учитывая тот факт, что язык программирования Perl изначально предназначался для обработки текста, можно прийти к выводу, что Perl и XML образует весьма удачную комбинацию. В этой ситуации возникает вполне логичный вопрос: каким образом следует сочетать эти компоненты? Именно на него и дает ответ наша книга.

Для кого написана эта книга

Книга предназначается для программистов, интересующихся вопросами применения языка Perl для обработки XML-документов. Предполагается, что читатель уже знаком с основами Perl. В ином случае советуем вам обратиться к книге В. Водолазкою «Энциклопедия Perl»[1].

При чтении книги не требуется большого опыта работы с XML. Но восприятие материала значительно упростится, если вы будете знакомы с какими-нибудь языками разметки, например с HTML. В случае необходимости обратитесь к книге А. Гончарова «Самоучитель HTML»[2].

При изучении материала книги крайне желательно иметь доступ к Интернету, а особенно — к всеобъемлющей архивной сети Perl (CPAN, Comprehensive Perl Archive Network), поскольку по мере чтения вам придется загружать те или иные модули из сети CPAN.

И конечно, гарантом успешного освоения материала книги будет ваше самостоятельное программирование на Perl и XML. Хотя эта книга ограничена по объему, но содержащихся в пей сведений вполне достаточно для успешного начала работы.

Структура книги

Книга состоит из десяти глав.

Глава 1 носит вводный характер. Нетерпеливые читатели книги здесь же могут познакомиться с первым практическим примером.

Глава 2 предназначена для читателей, которые лишь поверхностно знакомы с XML. Здесь находится синтаксический справочник, включающий описание структуры и форм выражения этого языка. Если вы хорошо знакомы с XML, можете смело пропустить эту главу (только потом не жалуйтесь на то, что не можете понять дальнейшего материала).

В главе 3 демонстрируются способы чтения и записи данных при работе с XML-документом. Конечно, самое интересное происходит на других этапах, но в любом случае будет полезным знать, как следует выполнять эти основные операции.

Глава 4 посвящена описанию потоков событий, образующих ядро большинства операций по обработке кода XML.

В главе 5 описываются основы Simple API, предназначенного для обработки XML-документов (SAX), — стандартного интерфейса для потоков событий.

Глава 6 посвящена вопросам обработки деревьев, образующих базовую структуру всех XML-документов. Мы начнем с простых структур встроенных типов данных и постепенно перейдем к усложненным объектно-ориентированным моделям деревьев.

В главе 7 рассматривается объектная модель документа (Document Object Model, DOM) — еще один стандартный распространенный интерфейс. Здесь же приводятся примеры, демонстрирующие ускорение обхода XML-деревьев средствами DOM.

Глава 8 посвящена дополнительным вопросам обработки деревьев, включая деревья событий и сценарии преобразования.

В главе 9 рассматриваются приложения, созданные с помощью Perl и XML, реально применяемые в повседневной практике.

---

[1] *Водолазкий В.* Энциклопедия Perl. — СПб.: Питер, 2001.

[2] *Гончаров А.* Самоучитель HTML. — СПб.: Питер, 2001.

Глава 10 подводит итог изученному материалу. Теперь, когда освоены модули, рассматривается их применение на практике, а также отмечаются возможные «подводные камни».

Источники информации

Несмотря на то что в книге содержится весь материал, необходимый для начала программирования на Perl и XML, со временем модули изменяются, появляются новые стандарты, в результате чего у вас может возникнуть подозрение относительно неполноты излагаемого материала. Эту проблему можно решить, используя дополнительные источники информации.

Список рассылки perl-xml

Если вы планируете работать с Perl и XML, используя все открывающиеся при этом возможности, первым делом изучите этот список рассылки. Для того чтобы подписаться на него либо просмотреть статьи из прошлых выпусков, необходимо обратиться к web-узлу http://aspn.activestate.com/ASPN/Mail/Browse/Threaded/perl-xml.

Можно также обратиться к web-узлу по адресу http://www.xmlperl.com, содержащему сведения по всем вопросам, относящимся к программированию на Perl/XML.

Сеть CPAN

Большинство модулей, рассматриваемых в книге, не включены в комплект поставки Perl, поэтому вам потребуется загрузить их из сети CPAN. Если вы уже работали ранее с Perl, то, наверное, знакомы с этой сетью. В этом случае процесс загрузки и установки модулей не составит особого труда. Если же вы не обладаете опытом подобного рода, обратитесь к web-узлу по адресу http://www.span.org. На этом узле найдите раздел часто задаваемых вопросов и ответов (FAQ), где содержится необходимая справочная информация. Ознакомившись с этим разделом, найдите требуемый Perl-модуль (имейте в виду, что подобный модуль может включаться в стандартный Perl-дистрибутив).

Благодарности

Авторы книги благодарны Поле Фергюссон (Paula Ferguson) за ее грамотное руководство. Они также признательны Энди Ораму (Andy Oram), Джону Орванту (Jon Orwant), Мишелю Родритесу (Michel Rodriguez), Саймону Сент-Лорену (Simon St.Laurent), Мэтту Сержанту (Matt Sergeant), Илье Стерину (Ilya Sterin), Майку Стоку (Mike Stok), Нэту Торкингтону (Nat Torkington), а также их редактору, Линде Мюи (Linda Mui).

Эрик благодарен своей жене Дженине; своей семье (Бриджит, Хелен, Эду, Антону, Элу, Джон-Полю, Джону и Мишель, Джону и Долорес, Джиму и Джоанне, Жину и Маргарет, Лиане, Тиму и Донне, Терезе, Кристоферу, Мэри-Энн, Анне, Тони, Полю и Шерри, Лиллиан, Бобу, Джою и Пэм, Элайне и Стиву, Дженифер и Мэрион); своим замечательным друзьям Деррику Эрнелю (Derrick Arnelle), Стаси Чандлер (Stacy Chandler), Дж. Д. Куррану (J. D. Curran), Саре Демб (Sarah Demb), Райану Фразье (Ryan Frasier), Крису Джернону (Chris Gernon), Джону Григсби (John Grigsby), Эндт Гроссеру (Andy Grosser), Лизе Музикер (Lisa Musiker), Бену Салтеру (Benn Salter), Кэролайн Сэнай (Caroline Senay), Грегу Тревису (Greg Travis), Барбаре Юнг (Barbara Young); а также своим сотрудникам: Ленин, Мэле, Нэйлу, Майку и Шэрил.

Джейсон хотел бы выразить признательность Джулии за то, что она всячески поддерживала его при работе над этим проектом; играм от фирмы Looney Labs (http://www.looneylabs.com), а также Бостону Уоррену (Boston Warren), поддерживающему здоровье автора напоминаниями о необходимости отдыха; Джошу и Османской империи, которые позволяли ему иногда уходить от реальности; кафе Diesel Cafe в Соммервилле (штат Массачусетс) и кафе 1369 Coffee House в Кембридже, которые совершенно случайно сыграли роль его альтернативных офисов; соседям Чарли, Кэри и сериалу: The Cat; фирме Apple Computer за превосходный ноутбук iBook и Mac OS X, которые чаще всего применялись при написании книги и выполнении хакерских упражнений; ну и конечно, Ларри Уоллу (Larry Wall) и всей его чудесной и восхитительной команде, которые подарили нам Perl.

От издательства

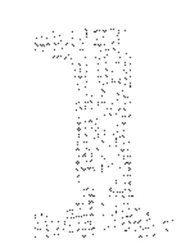
Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

На web-узле издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Perl и XML



Язык программирования Perl появился достаточно давно и изначально был ориентирован па обработку текста. В отличие от Perl, за «плечами» XML — всего лишь считанные годы, но за это время он успел зарекомендовать себя с самой лучшей стороны. Этот язык широко применяется для обработки web-содержимого, выполнения различных операций с документами, проектирования web-служб, а также в любой другой ситуации, в которой требуется структурирование изменчивой информации. Несмотря на столь различную природу, эти языки прекрасно уживаются вместе. И именно описанию истории их порой сложного, но удачного союза посвящена эта книга.

В чем причина тесного союза между Perl и XML?

В первую очередь следует отметить, что Perl идеально подходит для обработки текста. Он поддерживает дескрипторы файлов, объекты документов, выполняет обработку строк, а также предлагает возможности по созданию регулярных выражений. Любой, кому приходилось изначально разрабатывать программы па низкоуровневом языке, таком как С, а потом — на Perl, понимает, что Perl гораздо лучше приспособлен для обработки текста. А XML, по существу, — это обычный текст, поэтому Perl и XML прекрасно дополняют друг друга.

Более того, начиная с версии 5.6, Perl допускает работу с символьными кодировками, основанными на Unicode (например, UTF-8), и именно этот факт чрезвычайно важен при обработке XML-документов. Дополнительные сведения о символьных кодировках можно найти в главе 3.

Во-вторых, следует так же учитывать наличие всеобъемлющей архивной сети Perl (CPAN), включающей множество Perl-модулей, доступных всем желающим. Благодаря этому задачи программиста значительно упрощаются; каждый, начинающий программировать на языке Perl, может просто воспользоваться готовыми модулями. Благодаря этому достигается значительная экономия времени и средств. Например, зачем создавать собственный анализатор кода (parser), если в сети CPAN

содержится множество готовых анализаторов, доступных для свободной загрузки? Причем, как правило, все модули заранее протестированы и могут подстраиваться под конкретные нужды. Сеть CPAN не относится к числу жестко заданных структур: в ее развитие вносит вклад множество людей, а всякие проявления контроля имеют ограниченный характер. Как только возникает новая технология, в сети CPAN появляется поддерживающий ее модуль. Благодаря этому свойству прекрасно дополняются возможности XML, в результате чего могут изменяться старые и добавляться новые вспомогательные технологии.

Изначально XML-модули росли и множились «как грибы после дождя». Каждый модуль был снабжен уникальным интерфейсом и имел присущий ему оригинальный стиль в традициях Perl. В последнее время начала проявляться тенденция к созданию универсального интерфейса, позволяющего реализовать взаимозаменяемость модулей. Если вас в силу каких-либо причин не устраивает синтаксический анализатор SAX, вы можете легко воспользоваться каким-либо другим анализатором, не прилагая для этого дополнительных усилий.

В-третьих, гибкие возможности Perl, обеспечивающие объектно-ориентированное программирование, являются весьма полезными при работе с XML. Данные в XML-документе имеют иерархическую структуру, образованную с помощью базовых единиц. Эти единицы называются XML-элементами и могут содержать вложенные элементы. В результате элементы, образующие документ, могут быть представлены с помощью одного класса объектов, включающего простые идентичные интерфейсы. Более того, язык разметки XML инкапсулирует содержимое этих объектов , которые, в свою очередь, инкапсулируют код и данные. В результате они прекрасно дополняют друг друга. Нетрудно заметить, что подобные объекты весьма полезны при организации модульной структуры XML-процессоров. Они включают анализаторы, фабрики анализаторов, вспомогательные объекты, а также анализаторы, возвращающие объекты. Все это обеспечивает создание «прозрачного» кода, который может выполняться практически на всех платформах.

В-четвертых, большое значение имеет связь между Perl и Web. Конечно, Java и JavaScript обладают поистине неисчерпаемыми возможностями в этой области, но любой, кто немного разбирается в web-программировании, скажет вам, что Perl применяется при организации серверной части большинства web-серверов. Многие web-библиотеки, написанные на языке Perl, могут легко адаптироваться с учетом их применения в XML. Опытные программисты, которые годами разрабатывали web-узлы на языке Perl, могут свободно стать подданными «королевства» XML.

И наконец, при выборе языка программирования следует исходить из личных мотивов. Язык Perl идеален при работе с XML-кодом, но не следует замыкаться исключительно па нем. Просто попробуйте поработать с ним.

XML проще, чем вы думаете

Многие люди склонны рассматривать XML как результат вмешательства некоего «злого гения», который, как минимум, собирается уничтожить все человечество. Внедренный язык разметки, с его угловыми скобками и слэшами, на первый взгляд кажется довольно сложным. К тому же, если учитывать наличие вложенных элементов, типов узлов и объявлений DTD, ситуация значительно осложняется.

А теперь авторы книги хотят поделиться с читателем маленьким секретом: создание программ, предназначенных для обработки XML-кода, не составляет осо-

бого труда. Существует целый спектр инструментальных средств, ориентированных па выполнение синтаксического анализа и построения структур данных. Причем в распоряжение пользователя предоставляются удобные в применении API, овладеть которыми возможно в течение нескольких минут. Если вы хотите испытать всю прелесть по-настоящему сложных XML-приложений, особых препятствий для этого нет, но все же не следует чрезмерно все усложнять. Сложность XML-кода варьируется в очень широких пределах, и в случае разработки простого XML-приложения следует пользоваться простыми инструментальными средствами.

Для иллюстрации справедливости этого утверждения рассмотрим простой базовый модуль, именуемый `XML::Simple`, созданный Грантом Мак Клином (Grant McLean). При минимальных трудозатратах со стороны пользователя обеспечивается доступ к солидному набору полезных свойств, обеспечивающих обработку XML-кода.

Как известно, типичная программа считывает XML-документ, выполняет некоторые изменения, записывая их при этом в файле. Назначение модуля `XML::Simple` заключается во всемерной автоматизации этого процесса. В результате вызова подпрограммы происходит считывание XML-документа и сохранение его в памяти. Для представления элементов и данных применяются встроенные хэш-символы. После завершения всех необходимых изменений вызывается другая подпрограмма, выполняющая запись информации в файл.

А теперь перейдем к непосредственным практическим испытаниям. Как и в случае с любым другим модулем, директива use применяется для объявления `XML::Simple` в программе:

```
use XML::Simple;
```

После выполнения этой инструкции `XML::Simple` в пространство имен экспортирует две подпрограммы:

- `XMLin()` — эта подпрограмма считывает XML-документ из файла или строки и создает структуру, включающую данные и элементы. В ходе осуществления этого процесса возвращается ссылка на хэш, содержащий структуру.
- `XMLout()` — эта подпрограмма, располагая ссылкой на хэш с закодированным документом, генерирует XML-разметку и возвращает ее в виде строки текста.

При желании можно создавать документ «с нуля» путем генерирования структур данных на основе хэш-символов, массивов и строк. Применение подобного метода рекомендуется при первичном создании файла. Воздерживайтесь от применения круговых ссылок или некорректно функционирующих модулей.

Предположим, что ваш босс собирается отослать сообщения группе пользователей приложения WarbleSoft SpamChucker, управляющего списками рассылки. Одним из свойств этого приложения является возможность импорта/экспорта XML-файлов, представляющих списки рассылки. Единственная проблема в этом случае заключается в том, что боссу затруднительно читать на экране имена пользователей в их исходном виде и он предпочитает, чтобы они отображались с применением заглавных букв. В силу этого требуется написать программу, редактирующую XML-файлы данных и выполняющую требуемые преобразования.

Первой выполняемой задачей в этом случае будет анализ XML-файлов с целью определения стилей разметки. Образец подобного документа приводится в листинге 1.1.

Листинг 1.1. Файл данных SpamChucker

```
<?xml version="1.0"?>
<spam-document version="3.5" timestamp="2002-05-13 15:33:45">
<!- Автоматически генерируется WarbleSoft Spam, версия 3.5 ->
<customer>
     <first-name>Joe</first-name>
     <surname>Wrigley</surname>
     <address>
          <street>17 Beable Ave . </street>
          <sity>Meatball</city>
          <state>MI</state>
          <zip>82649</zip>
     </address>
     <email>joewrigley@jmac.org</email>
     <age>42</age>
</customer>
<customer>
     <first-name>Henrietta</first-name>
     <surname>Pussycat</surname>
          <address>
          <street>R.F.D. 2</street>
          <city>Flangerville</city>
          <state>NY</state>
          <zip>83642</zip>
          </address>
     <email>meow@263A.org</email>
     <age>37</age>
</customer>
</spam-document>
```

Ознакомившись со страницей perldoc, описывающей модуль XML : : Simple, вы сможете чувствовать себя достаточно уверенно и написать небольшой сценарий, показанный в листинге 1.2.

Листинг 1.2. Сценарий, выполняющий замену строчных на прописные буквы в именах заказчиков

```
# Программа выполняет замену строчных букв на прописные
# в именах заказчиков по всему XML-документу,
# сформированному приложением WarbleSoft SpamChucker..
# Включение ограничений и предупреждений, в этом случае
# вы знаете что делать.
use strict;
use warnings;
# Импорт модуля  XML::Simple.
use XML::Simple;
# Включение файла в хэш-ссылку с помощью процедуры
# "XMLin" из модуля
XML::Simple.
# Также включается опция 'forcearray', поэтому
# все элементы
# содержат ссылки на массив.
my $cust_xml = XMLin('./customers.xml', forcearray=>l) ;
# Выполнение цикла по каждому субхэшу customer,
# причем все объекты хранятся в виде анонимного
# списка под ключом 'customer'.
 for my $customer (@{$cust_xml->{customer}}) {
```

```
        # Замена строчных на прописные буквы в элементах
        #'first-name' и 'surname' путем выполнения встроенной
        # функции Perl, uc().
        foreach (qw(first-name surname)) {
            $customer->{$_}->[0] = uc($customer->{$_}->[0]);
        }
    }

    # Повторный вывод на печать хэша в виде XML-документа,
    # включающего завершающий символ новой строки
    # (для улучшения восприятия кода) .
    print XMLout($cust_xml);
    print "\n";
```

В результате выполнения программы (возможно, связанного с некоторыми проблемами, с тех пор как всеми данными распоряжается ваш босс), получаем следующий результат:

```
<opt version="3.5" timestamp="2002-05-13 15:33:45">
    <customer>
        <address>
            <state>MI</state>
            <zip>82649</zip>
            <city>Meatball</city>
            <street>17 Beable Ave.</street>
        </address>
        <first-name>JOE</first-name>
        <email>i-like-cheese@jfflac.org</email>
        <surname>WRIGLEY</surname>
        <age>42</age>
    </customer>
    <customer>
        <address>
            <state>NY</state>
            <zip>83642</zip>
            <city>Flangerville</city>
            <street>R.F.O. 2</street>
        </address>
        <first-name>HENRIETTA</first-name>
        <email>ineowmeow@augh.org</email>
        <surname>PUSSYCAT</surname>
        <age>37</age>
    </customer>
</opt>
```

Примите наши поздравления! Вы написали программу, выполняющую обработку XML-кода, которая, к тому же, хорошо работает. Достигнут почти превосходный результат. Применяемое здесь слово «почти» свидетельствует о том, что поведение программы немного отличается от ожидаемого. Поскольку хэш-символы не сохраняют порядок следования содержащихся в них элементов, он будет изменен. Также могут отсутствовать пробелы между элементами. Может ли это представлять проблему?

В рассматриваемом сценарии подчеркивается важная мысль: необходимость компромисса между простотой и сложностью. Как разработчику вам потребуется принять решение относительно того, что составляет важную часть кода, написанного на языке разметки, а что — нет. Иногда имеет значение порядок следования элементов. В этом случае использование модулей, подобных XML : : Simple, не до-

пускается. Либо программисту требуется получить доступ к обрабатываемым инструкциям, а также сохранять их в файле. И снова модули, подобные `XML::Simple`, не обладают подобными возможностями. Поэтому прежде, чем принять решение относительно использования конкретного модуля, требуется оценить его возможности. В данном случае все значительно проще, поскольку вы проконсультировались со своим боссом и проверили программу SpamChucker, воспользовавшись набором измененных данных. В результате были удовлетворены все заинтересованные лица. Полученный и исходный документы похожи, благодаря чему можно прийти к выводу о соответствии требованиям, выдвигаемым к приложению[1]. Поэтому считайте, что вы прошли «боевое крещение» и приступили к обработке XML-документа с помощью Perl!

Имейте в виду, что мы находимся только в начале пути. Большая часть книги оформлена в виде курса опережающего обучения и содержит множество советов и методик, предназначенных для выполнения обработки любых XML-документов. Не каждая проблема, имеющая отношение к этому процессу, является столь же простой, как указывалось выше. В любом случае авторы книги надеются, что читатели достаточно подготовлены и не испытывают непреодолимых сложностей, связанных с обработкой XML-документов средствами Perl.

XML-процессоры

Теперь, когда была рассмотрена «простая сторона» XML, приступим к изучению некоторых особенностей этого языка. Эти особенности следует учитывать при работе с XML и Perl.

В книге часто встречается термин *XML-процессор* (нередко сокращаемый до слова *процессор,* который в корне отличается от соответствующего термина, обозначающего центральное вычислительное устройство компьютерной системы). Этому термину присущ в максимальной степени обобщенный смысл. Для процессора не суть важно, что именно делает программа с обрабатываемым им документом. Он не определяет источники происхождения документа, а также методы его дальнейшей обработки.

Как и следовало ожидать, функционирование XML-процессора, рассматриваемое в чистом виде, не представляет особого интереса. Исходя из этих соображений, можно прийти к выводу, что компьютерная программа, которая делает что-либо полезное при обработке XML-документа, использует процессор просто в качестве одного из компонентов. Процессор обычно считывает XML-файл, а затем, применяя возможности синтаксического анализа, преобразует его в структуры, находящиеся в памяти, которые далее обрабатываются программой.

В мире Perl подобное поведение определяется с помощью Perl-модулей: как правило, в случае, когда требуется обработка XML-кода, осуществляемая с помощью инструкции `use`. При этом используется существующий пакет, обеспечиваю-

---

[1] Иногда можно сказать, что, несмотря на некоторые имеющиеся различия, оба документа являются *семантически эквивалентными,* хотя это утверждение не вполне соответствует действительности. Порядок следования элементов был изменен, и подобное обстоятельство является весьма существенным в XML. Поэтому в данном случае можно утверждать, что документы совпадают настолько, что отвечают всем требованиям относящегося к ним программного обеспечения, а также конечного пользователя.

щий доступ программиста к объектно-ориентированному интерфейсу. Начало обработки XML-кода во многих программах на языке Perl, располагающих соответствующими возможностями, определяется с помощью анализатора XML: : Parser (либо другой подобной программы). По прошествии небольшого периода времени вся «черновая работа» по разбору XML-кода поручается другим, ранее написанным модулям. Код, составленный программистами, определяет порядок предварительной и завершающей обработки.

Пользуйтесь готовыми модулями

К одной из сильных сторон Perl можно отнести то, что этот язык поддерживается сообществом его сторонников во всем мире. Как только программисты на языке Perl идентифицируют проблему и создают соответствующий модуль, нацеленный на ее решение, результат их деятельности становится достоянием мировой общественности. Для этой цели предназначается сеть CPAN. Основное преимущество в данном случае заключается в том, что если вы хотите создать какой-либо фрагмент кода на языке Perl, существует вероятность, что кто-либо создал его ранее, и в сети CPAN можно найти соответствующий Perl-модуль.

Однако метод «коллективного творчества», примененный по отношению к столь «юной», популярной и нетрадиционной технологии, каковой является XML, имеет свои недостатки. К моменту первого «выхода на сцену» XML в сети CPAN существовали различные Perl-модули, написанные различными программистами. По причине полной анархии все они образуют «нестройный хор», в число участников которого входят различные структуры и интерфейсы, ориентированные на достижение различных целей.

Однако не падайте духом. Времена «анархии и беспорядка», относящиеся к 1998 году, остались в прошлом. В настоящие время наблюдается некое подобие организации и стандартов. Причем инициатива исходит от сообщества Perl/XML (о чем изначально было заявлено в списке рассылки *perl-xml*, поддерживаемом ActiveState). Члены сообщества разрабатывали первые модули с целью создания требуемых инструментов. При этом они следовали правилам, установленным другими игроками в мире XML. К числу этих «игроков» можно отнести стандарты синтаксического анализа, SAX и DOM, а также внедренные XML-технологии, такие как XPath. Позднее появились базовые анализаторы низкого уровня. Совсем недавно возникли интересные системы (такие как XML: : SAX), которые реализуют модель DWIM на уровне Perl, отображенную в разрабатываемых стандартах[1].

Конечно, если вы желаете воспользоваться «бестолковыми» инструментами, пригодными лишь для выполнения черновой и быстрой работы, то всегда сможете это сделать. К числу таких инструментов можно отнести модуль `XML: : Simple`. Авторы книги приложат максимум стараний для того, чтобы помочь вам воспользоваться стандартизованными инструментами. После этого вам останется лишь запустить процесс обработки XML-кода и не вмешиваться в происходящий процесс.

[1] Аббревиатура DWIM расшифровывается как «Do What I Mean» («Делать то, что задумано»), это один из фундаментальных принципов, заложенных в основу Perl.

Программисту на заметку

Как правило, XML-модули, находящиеся в сети CPAN, удовлетворяют потребности программистов на 90%. Конечно, оставшиеся 10% можно рассматривать как соотношение между ведущими специалистами вашей компании и «кандидатами на увольнение». Авторы книги намереваются оправдать те затраты, которые вы понесли в результате приобретения книги, путем демонстрации некоторых «ужасных» деталей, объясняющих порядок обработки в Perl XML-документов на самых низких уровнях (по сравнению с любым видом специализированной обработки текста, осуществляемой в Perl). Для начала обратимся к некоторым «азбучным истинам», которые следует учитывать в дальнейшем.

Происхождение программы не имеет значения

К тому времени, когда часть XML-кода, выполняющая анализ XML-документов, приступает к его обработке, источник его происхождения не имеет особого значения. Документ может быть получен с помощью локальной сети, загружен с базы данных либо считан с диска. Для анализатора важен только XML-код и все, что с ним связано.

Имейте в виду, что программа требует внимания к себе в целом. Например, если была написана программа, реализующая механизм XML-RPC, лучше знать порядок использования протокола TCP, который определяет выборку и отсылку через Интернет всех XML-данных! Мы можем использовать эту программу для выполнения операций выборки и отсылки данных, однако до тех пор, пока конечный программный продукт является неизменным, так как пользователь будет испытывать потребность в чистом XML-документе, который может обрабатываться XML-процессором, лежащим в ядре программы.

Все XML-документы подобны с точки зрения структуры

Независимо от цели и способа создания, при составлении любого XML-документа должны учитываться одни и те же базовые правила форматирования: строго один корневой элемент, отсутствие перекрывающихся элементов, заключение всех атрибутов в кавычки и т.д. Каждый компонент анализатора для XML-процессора испытывает потребность в выполнении тех же самых операций, что и любой другой XML-процессор. Это в свою очередь означает, что все процессоры могут совместно использовать общую базу. В программах на языке Perl, выполняющих обработку XML-кода, обычно используются свободно распространяемые модули-анализаторы. Практика повторной реализации базовых XML-процедур анализа не применяется.

Более того, благодаря однодокументной природе XML, процесс обработки кода превращается в приятный итеративный процесс, не требующий больших затрат времени. При этом каждый документ, посредством внешней сущности другого документа, испытывает «магическое превращение» в «просто другой элемент» в составе вызывающего процесса. В этом случае код, который образует первый документ, может создавать«ткань» любой ссылки (либо какого-либо другого объекта, к которому может иметь отношение ссылка), не требуя дополнительных усилий от программиста.

XML-приложения различаются своим назначением

Любые XML-приложения определяют смысл существования произвольного XML-документа. Причем набор правил высшего уровня, которым следует произвольный XML-документ, способствует достижению некоторых полезных целей. Эти правила могут определять: заполнение файла конфигурации, подготовку передачи данных в сети иливыполнение некоторых других действий. Смысл существования XML-приложений заключается не только в том, чтобы наполнять скромные документы «высшим смыслом их предназначения». Они также требуются для определения структуры создаваемых документов в соответствии с определенной спецификацией приложений. С помощью объявления DTD облегчается достижение совместимости описанной выше структуры. Однако следует учитывать тот факт, что в распоряжении разработчика может не оказаться схемы формального подтверждения, используемой при разработке приложений. Может возникнуть потребность в создании некоторых правил проверки. Эти правила окажутся весьма полезными, например, в случае, когда требуется, чтобы ваши последователи (включая и вас самих две недели спустя) не «путались в дебрях» разработанной ранее программы. Потребуется также создать схему проверки, если необходимо будет позволить другим программистам создавать программы, обеспечивающие использование преимуществ языка XML.

Как правило, при реализации на практике большинства приемов составления XML-кода, во главу угла ставится упомянутый выше дуализм «документ/приложение». В большинстве случаев разрабатываемое программное обеспечение будет включать разделы, учитывающие три перечисленных ниже факта:

- ввод данных осуществляется с применением подходящего метода. В частности может осуществляться «прослушивание» сетевого сокета либо считывание файла с диска. Подобное поведение является весьма типичным и характерным для Perl: делайте все необходимое для получения данных;
- перехваченные входные данные будут передаваться некоему типу XML-процессора. Как правило, лучше всего воспользоваться одним из анализаторов, созданным и поддерживаемым сообществом разработчиков на Perl. В качестве этих модулей может использоваться модуль `XML::Simple` либо более сложные модули, которые будут рассмотрены ниже;
- и наконец, обратите внимание на результат обработки процессором XML-кода. Возможно, он будет продуцировать XML-код (либо HTML-код), обновлять базу данных либо отсылать электронное сообщение вашей матери. Этот пункт является определяющим при выполнении XML-приложения: просто берется код XML и выполняется некая его обработка. В книге не будут обсуждаться поистине безграничные возможности, открывающиеся в этом случае. Предмет рассмотрения составят тесные связи между XML-процессором и остальными частями вашей программы.

Особенности XML

В этом разделе затрагиваются вопросы, составляющие предмет всей книги. Именно с ними связаны проблемы, возникающие при обработке XML-документов.

Формальная корректность

В XML имеется встроенная система контроля качества. Документ должен соответствовать некоторому минимальному набору синтаксических правил, позволяющих соответствовать формальной корректности для XML-кода. Большинство анализаторов не способны обрабатывать документ, который нарушает любое из этих правил, поэтому требуется убедиться в том, что произвольные входные данные характеризуются приемлемым уровнем качества.

Кодировки символов

Жизнь в XXI веке требует уделять внимание таким вопросам, как применяемые кодировки символов. Безвозвратно ушли те дни, когда содержимое web-узловв Интернете кодировалось с применением набора символов ASCII. «Героем наших дней» стал Unicode, па базе которого формируются все основные наборы символов, применяемые в Сети. В XML предполагается работа с символами Unicode, хотя существует множество способов для представления этой кодировки, включая наиболее часто используемую в Perl кодировку Unicode, UTF-8. Как правило, достаточно редко приходится задумываться о вопросах подобного рода, по нужно быть осведомленным относительно имеющихся возможностей.

Пространства имен

Не каждый может похвалиться тем, что работал с пространствами имен. Пространства имен производят разбиение кода па отдельные области, разделяя теги, разметки и объявления. В результате появляется возможность смешивать и сопоставлять различные типы документов. Идет ли речь об уравнениях в HTML-коде, либо о разметке в виде данных в XSLT-коде, использование пространств имен является оправданным. Имейте в виду, что поддержка пространств имен реализована в недавно разработанных модулях.

Объявления

По существу, объявления не входят в состав документа, а просто описывают его. Это следует принимать как данность и не стоит уделять этому вопросу повышенное внимание. Помните лишь о том, что в документах часто применяются объявления DTD, а также включаются объявления таких объектов, как сущности и атрибуты. Следует учитывать это, с тем чтобы не «наломать дров» в дальнейшем.

Сущности

Сущности и взаимосвязи сущностей выглядят достаточно просто: они остаются в содержимом, которое вы, скорее всего еще не определили. Возможно, что содержимое находится в другом файле, либо включает символы, ввод которых вызывает определенные затруднения. Иногда требуется разрешить ссылки, а иногда лучше воздержаться от этого. Норой анализатор выполняет просмотр объявлений, а в другое время он не заботится об этом. Сущности могут содержать другие сущности, причем глубина вложенности не ограничивается. В любом случае нужно контролировать ситуацию во избежание возможных проблем в дальнейшем.

Служебные символы

Согласно правилам, принятым в XML, все то, что не является тегом разметки, относится к разряду значащих символьных данных. Этот факт может привести к некоторым неожиданным результатам. Например, не всегда можно определенно сказать, что происходит при обработке служебных символов. По умолчанию XML-процессор сохраняет все символы — даже символы создания новых строк, которые можно включать после тегов с целью улучшения восприятия кода, либо символы пробелов, которые можно применять для создания отступов в тексте. Некоторые анализаторы иногда позволяют игнорировать пробелы, но в этом случае отсутствуют жестко установленные и простые правила.

Заканчивая обзор языков Perl и XML, можно сделать вывод, что они превосходно дополняют друг друга. И хотя в процессе работы могут появляться так называемые «ловушки», но благодаря наличию различных модулей, разработанных программистами, изучение возможностей Perl/XML будет легким и приятным.

Краткий курс XML



Язык XML — результат длительной эволюции, в ходе которой происходил «естественный отбор» лучших качеств, присущих языкам разметки предыдущего поколения. Процесс эволюционных изменений привел к качественному скачку, следствием которого стало объединение преимуществ обобщенного языка разметки SGML, простоты разметки свободной формы, а также правил формальной корректности. Однозначная структура и интуитивно понятный синтаксис значительно облегчают процесс обработки XML-кода другими программами.

На базе XML пользователь может создавать собственный язык разметки, лучше приспособленный для обработки различных данных. В частности, стандартные теги могут заменяться сконструированными пользователем элементами. При необходимости можно выполнить формализацию языка с помощью объявлений элементов и атрибутов, указанных в определении типа документа (Document Type Description, DTD).

Основные компоненты синтаксической структуры XML будут следующими: сущности, комментарии, инструкции по обработке данных, а также секции CDATA. С их помощью выполняется группировка элементов и атрибутов в соответствии с различными пространствами имен, а также формирование словаря пользовательских документов. Так, например, атрибут `xml:space` позволяет варьировать величину пробелов в тексте. Это обстоятельство может играть решающую роль при разметке, когда восприятие кода человеком является столь же важным, как и корректное форматирование.

Поддержка и модификация документов значительно облегчается при использовании некоторых полезных технологий. Достоверность XML-конструкций, устанавливаемая на основе канонической модели, проверяется с помощью схем и объявлений DTD. Схемы предлагают пользователю богатый набор возможностей. В частности, с их помощью поддерживаются шаблоны на основе символьных данных, а также обеспечивается улучшенный синтаксис модели содержимого. Язык XSLT выполняет преобразование документов в различные формы и может применяться при обработке XML-документов. Причем этот способ обработки явля-

ется более простым, чем создание отдельной программы, хотя всякое правило имеет свои исключения.

В следующих разделах главы вы найдете обзор возможности"! XML, краткий исторический очерк, а также описание структуры и методов применения этого языка.

XML: краткий исторический очерк

В прежние годы обработка текста жестко привязывалась к аппаратным средствам, с помощью которых выполняется его отображение (мониторы). Сложное форматирование текста также напрямую зависело от применяемых принтеров.

В качестве примера рассмотрим язык troff. Этот язык форматирования текста в свое время был весьма популярным и включался в состав большинства дистрибутивов Unix. Он позволял выполнять высококачественное форматирование текста, которое ранее было возможным только с применением полиграфических машин.

Текст, обрабатываемый редактором troff, представлял собой смесь инструкций и данных. В качестве инструкций применялись сочетания символов, предваряемые обратной косой чертой. Так, например, набор символов \fI изменял текущий стиль шрифта па курсив. При отсутствии символа обратной косой черты оставшиеся символы воспринимались в качестве данных. Подобный смешанный набор инструкций и данных именуется *разметкой.*

Язык troffобладает и другими, более «топкими» возможностями. Так, например, инструкция .vs 18p указывает программе-форматтеру на необходимость установки межстрочного интервала, равного 18-ти пунктам. Если не учитывать эстетические соображения, довольно затруднительно указать назначение да иного интервала. Можно лишь отметить, что подобный интервал определяет специфичную инструкцию для процессора, которая не может быть интерпретирована каким-либо другим образом. Поэтому такая инструкция может быть полезной только при подготовке документа к печати с применением определенного стиля. Если же в дальнейшем потребуется выполнить какие-либо изменения в документе, лучше ее не использовать.

А теперь представите себе ситуацию, когда возможности troffиспользуются при подготовке книги к изданию, причем каждый новый термин выделяется полужирным стилем. В этом случае документ может содержат!) несколько тысяч инструкций, определяющих полужирный стиль текста. Вы испытываете чувство глубокого удовлетворения от проделанной работы и уже собираетесь выводить макет книги па печать, как вдруг все изменяется. Сотрудник отдела верстки сообщает вам пренеприятную новость об изменении макета книга, вследствие чего все новые термины теперь должны выделяться курсивом. В результате появляется серьезная проблема, заключающаяся в необходимости преобразования каждой инструкции, определяющей полужирный стиль текста, в инструкцию, задающую курсивный стиль текста.

Первое, что приходит в голову в подобной ситуации, — открыть документ в окне редактора и выполнить операцию поиска и замены. Но, к своему ужасу, вы обнаруживаете, что полужирным стилем выделены не только новые термины. Этот стиль также применялся для выделений определенных мест в тексте, а также для обозначения наименований. В результате выполнения глобальной замены подобные выделения могут быть утеряны, причем самым непредсказуемым образом.

Поэтому возможно лишь последовательное изменение инструкций, на выполнение которого может потребоваться не один рабочий день.

Независимо от степени «разумности» языка форматирования, подобного troff, возможно повторение одной и той же проблемы. Причина этого заключается в том, что этот язык является репрезентативным. В *репрезентативном* языке разметки описание содержимого производится с применением терминов, определяющих форматирование текста. В языке troff определяются детали, характеризующие шрифты и интервалы, но не указывается содержательная информация. В результате применения редактора troff сужается спектр возможностей по дальнейшей обработке документа. Например, довольно затруднительной будет организация поиска с переходом к последнему параграфу третьего раздела документа. Вообще говоря, репрезентативная разметка затрудняет выполнение любой задачи, за исключением форматирования документов с целью их дальнейшего вывода на печать (именно для этого она и применяется в первую очередь).

Редактор troff может характеризоваться как средство выполнения *целевого форматирования*. Причем это средство не является универсальным, а может лишь применяться в ограниченных масштабах. Какой же формат может применяться в подобной ситуации? Будет ли использован «исходный» формат, который на самом деле не определяет какого-либо форматирования, а просто производит упаковку данных с помощью некоторого распространенного метода? Подобные вопросы стали возникать в конце 60-х годов прошлого столетия. Именно в это время была разработана концепция *обобщенного кодирования*. Суть этой концепции заключалась в разметке содержимого с помощью стиля отличного от репрезентативного. При этом, вместо инструкций но форматированию предусматривалось применение описательных тегов.

Первый практический проект в этой области был разработан в Ассоциации по координированию разработок в области графики (Graphic Communications Association, GCA). Этот проект назывался GenCode. В рамках проекта были разработаны методы кодирования документов с применением обобщенных тегов, а также сборки документов на основе нескольких элементов, являющихся «предшественниками» гипертекста. На базе этой концепции и под руководством фирмы IBM был создан обобщенный язык разметки (Generalized Markup Language, GML). Почетное право разработки этого языка принадлежало Чарльзу Голдфарбу (Charles Goldfarb), Эдварду Мошеру (Edward Mosher) и Реймонду Лори (Raymond Lorie)[1]. В результате завершения этой работы фирма IBM получила возможность выполнять операции по редактированию, просмотру, поиску и печати одного и того же исходного документа с помощью различных программ. Преимущества подобного решения становятся еще более очевидными в случае, когда объем документации той или иной компании приближается к миллионам страниц в год.

Голдфарб руководил группой по разработке стандартов в Американском национальном институте по разработке стандартов (American National Standards Institute, ANSI), прилагая все усилия для популяризации возможностей языка GML. На основе проектов GML и GenCode комитет ANSI разработал стандартный обобщенный язык разметки (Standard Generalized Markup Language, SGML). Сразу же после разработки этот язык был адаптирован Министерством обороны США (U.S Department of Defense) и Управлением по контролю внутренних доходов (Internal Revenue Service). Благодаря столь серьезному покровительству язык был обречен на неминуемый успех. В 1986 году он сертифицируется ISO в качестве международного стандарта.

[1] Интересно, что инициалы разработчиков совпадают с названием языка разметки (GML)

С того времени па базе этого стандарта было разработано множество инструментальных средств и пакетов, предназначенных для выполнения обработки текста.

Появление обобщенного кодирования в корне изменило понимание смысла цифрового содержимого, которое может быть описано путем указания его сути, а не способа отображения на экране. Так, следующий код скорее имеет отношение к базе данных, чем к файлу текстового процессора:

```
<personnel-record>
    <name>
        <first>Rita</first>
        <last>Book</last>
    </name>
    <birthday>
        <year>1969</year>
        <month>4</month>
        <day>23</day>
    </birthday>
</personnel-record>
```

Обратите внимание на отсутствие репрезентативной информации. Вы можете форматировать отображаемые данные произвольным образом: имя-фамилия, фамилия-имя либо разделять их запятой. Даты могут отображаться в американском (4/23/1969) либо европейском (23/4/1969) форматах. Для выполнения последнего действия потребуется просто выбрать порядок следования элементов <month> и <day >. Причем в документе не задается жестко использование подобных элементов, в силу чего возможны некоторые вариации.

Несмотря на столь выдающиеся возможности, SGML никогда не пользовался популярностью в небольших фирмах, занимающихся разработкой программного обеспечения. Прерогатива его применения в первую очередь принадлежит крупным компаниям, что во многом объясняется дороговизной и объемностью программного обеспечения. Применение этого языка потребует наличие команды разработчиков, выполняющих работы по установке и конфигурированию среды разработки SGML. Также использование SGML связано с бюрократией, различного рода конфликтами и большим потреблением ресурсов. В силу перечисленных причин, SGML в его исходной форме является малопригодным в современных условиях.

В связи с вышесказанным у читателя может возникнуть резонный вопрос: «А что же можно сказать относительно HTML? Не правда ли, что HTML — это подмножество SGML?». Язык разметки HTML, который является «звездой» Интернета, а также прародителем гипертекста и «рабочей лошадкой» Сети, на самом деле — приложение SGML. Под словом «подмножество» в данном случае подразумевается то, что язык разметки построен на базе правил языка SGML. На самом деле SGML не является языком разметки, а представляет собой инструментарий, применяемый для создания пользовательских описательных языков разметки. На основе SGML разработаны другие языки разметки, например, языки кодирования технической документации, IRS-формы и некоторые другие языки.

Несмотря на большую популярность, языку HTML присущи некоторые ограничения. Он является компактным, но в то же время в недостаточной степени описательным языком. По своим возможностям этот язык скорее напоминает troff, чем DocBook либо другие SGML-приложения. Например, в состав HTML вклю-

чены теги <i> и <b>, с помощью которых производится изменение стиля шрифта без объяснения причин этого действия. По причине ограниченности возможностей HTML и присущей ему частичной репрезентативности, вряд ли можно поставить его в один ряд с SGML. При использовании HTML возможно лишь отображение содержимого в каком-либо местоположении, а возможности по проведению дополнительной обработки весьма ограничены.

В результате разработчики стандартов решили предпринять еще одну попытку, заключающуюся в достижении компромисса между описательными возможностями SGML и простотой HTML. Плодом этих усилий стал расширяемый язык разметки (Extensible Markup Language, XML). Литера 'X' представляет собой сокращение от слова «extensible» (расширяемый). Именно в наименовании заключается первое очевидное отличие от HTML. Второе и более важное отличие состоит в том, что XML-документы не ограничены «прокрустовым ложем» набора HTML-тегов. Вы можете расширять пространство имен тегов, делая его описательным в той мере, в какой пожелаете. Здесь проявляется совместимость XML с SGML.

С момента своего возникновения, XML достиг ошеломляющего успеха. В ходе эволюции этого языка появились связанные с ним программные продукты: XML-RPC, XHTML, SVG и DocBook XML. В комплект поставки XML включены некоторые компоненты, в том числе XSL (выполнение операций по форматированию), XSLT (выполнение преобразований), XPath (выполнение поиска) и XLink (выполнение компоновки). Большинство стандартов в этой области было разработано под руководством консорциума (World Wide Web Consortium, W3C). Членами этой уважаемой организации являются Microsoft, Sun, IBM, а также многие научные и общественные организации.

Основной задачей W3C являются исследования и поощрение развития новых технологий, применяемых в Интернете. Подобное утверждение может показаться абстрактным, поэтому если вы нуждаетесь в более конкретной информации, обратитесь на web-узел http://www.w3.org/. Консорциум W3C не занимается разработкой, контролированием либо лицензированием стандартов. Скорее он дает рекомендации, к которым следует прислушиваться разработчикам, но которые вовсе необязательны для выполнения[1].

В любом случае применяемая система остается в достаточной степени открытой, чтобы допустить возникновение конструктивных разногласий. К разногласиям подобного рода можно отнести недавние дискуссии и обсуждения, порожденные стандартом W3C, именуемым XML Schema. Этот случай более подробно будет рассмотрен в главе 3. Он является достаточно серьезным, но в то же время не следует преувеличивать связанную с ним возможную опасность. Отметим, что рекомендации консорциума W3C всегда доступны широкой общественности.

С того времени, как язык XML превратился в универсальное средство обработки данных, навыками работы с ним должны обладать все web-программисты. Оставшаяся часть главы представляет собой краткое введение в курс по языку XML.

---

[1] Если такая авторитетная организация, как W3C, дает какие-либо рекомендации, они часто приобретают силу закона; многие разработчики начинают следовать рекомендациям сразу же после их опубликования. Оставшейся части разработчиков также стоит придерживаться требований стандарта в случае, если они хотят обеспечить совместимость своих программ с разработанным ранее программным обеспечением.

Разметка, элементы и структура

Любой язык разметки обеспечивает метод встраивания инструкций в структуру данных. Благодаря этому облегчается процесс обработки данных компьютерными программами. В большинстве схем разметки, реализуемых я языках troff, TeX и HTML, предусмотрены инструкции, оптимизированные с учетом выполнения какой-либо одной задачи (например, форматирование документа с целью его вывода на принтер либо отображения на экране монитора). Подобные языки основываются на репрезентативном описании данных, предусматривающем контроль гарнитуры, цвета и размера шрифта, либо каких-либо других свойств, специфичных для данного шрифта. Хотя результатом применения такой разметки могут явиться прекрасно отформатированные документы, не стоит возлагать на нее слишком большие надежды. Ведь в этом случае вы можете навсегда «оказаться в плену» у одного формата, поскольку применение данных в других целях, возможно, будет весьма затруднительным.

Подобные проблемы предопределили появление XML. Этот обобщенный язык разметки описывает данные в соответствии с их структурой и предназначением. Репрезентативные сведения хранятся в специально выделенном местоположении (например, в таблице стилей). Благодаря функциональному описанию частей документа возможна его обработка с помощью разных инструментальных средств. Подобный документ носит универсальный характер и может применяться в самых различных целях.

А теперь приступим к рассмотрению базовых компонентов языка XML. Ключевым компонентом XML является *элемент.* По определению, элементами называются изолированные области данных, которые уникальны для данного документа. В качестве примера можно рассмотреть обычную книгу, состоящую из введения, глав, приложений и предметного указателя. Каждый из разделов книги маркируется в качестве уникального XML-документа. В свою очередь, элементы могут разбиваться на подэлементы (в любой книге есть названия глав, параграфы, примеры и отдельные разделы). Подобный процесс разбиения может включать несколько этапов (параграф включает такие элементы, как выделенный текст, цитаты либо гиперссылки).

Элементы выполняют функцию связывания меток и других свойств с данными. Каждому элементу сопоставлено имя (либо *тип элемента),* указывающее па его предназначение в документе. Элементу, выполняющему роль главы книги, может присваиваться имя «глава» (либо «гл» — в данном случае вы свободны в выборе). Также элемент может включать и другую информацию, например, пару «имя-значение», именуемую *атрибутом.* Сочетание типов и атрибутов элемента является уникальным отличительным признаком того либо иного элемента.

Листинг 2.1 представляет типичный XML-код.

Листинг 2.1. Фрагмент XML-кода

```
<list id="eriks-todo-47">
     <title>Things to Do This Week</title>
     <item>clean the aquarium</item>
     <item>mow the lawn</item>
     <item priority="important">save the whales</item>
</list>
```

Вы, наверное, уже догадались, что в примере определяется перечень задач, состоящий из трех элементов и заголовка. Каждый, кто когда-нибудь работал с HTML-кодом, сразу же узнает признаки разметки. Текстовые фрагменты, заключенные в угловые скобки ("<" и ">"), именуются *тегами.* В данном случае теги —

это разделители между элементами. Для каждого непустого элемента требуется указание начального и конечного тега, причем каждый тег содержит метку типа элемента. Начальный тег может дополнительно определять количество атрибутов (пары «имя-значение», такие как priori ty="important"). Как видите, разметка является достаточно «прозрачной» и однозначной даже на первый взгляд.

На самом деле очень важным является то, что подобная разметка может легко восприниматься компьютерными программами. Разработчики языка XML хорошо потрудились над тем, чтобы XML-документ мог легко обрабатываться всеми XML-процессорами, независимо от его содержимого либо типов применяемых тегов. Код такого документа однозначен, конечно, если придерживаться синтаксических правил. В результате значительно упрощается процесс обработки, поскольку не возникает ситуаций, требующих включения дополнительного кода.

А теперь рассмотрим HTML-код с точки зрения его первоначального определения (в качестве SGML-приложения, прародителя XML)[1]. При работе с некоторыми элементами можно не указывать конечный тег, поскольку, как правило, на основе содержимого можно заключить, где завершается элемент. Несмотря на это, за создание достаточно устойчивого кода, выполняющего обработку неоднозначных ситуаций, приходится платить. Плата, понесенная в этом случае, заключается в избыточной сложности и неточном выводе, формируемом на основе плохих гипотез. Попробуйте представить, что будет в случае, если тот же самый процессор может обрабатывать элемент любого типа, а не только HTML-элементы. Обобщенные XML-процессоры лишены возможности предсказания расположения отдельных элементов. Поэтому такие неоднозначные ситуации, как отсутствие завершающего тега, могут привести к появлению проблем.

XML-код может быть представлен в виде диаграммы, называемой *деревом*. Подобная структура знакома многим программистам. В верхней части дерева (информационные деревья растут в направлении «сверху-вниз») находится корневой элемент. Элементы, расположенные ниже, образуют ветви дерева. Каждый элемент может содержать другие элементы, относящиеся к другим уровням, и так до тех пор, пока не будет достигнута нижняя часть — «листья» дерева. Листья образуются либо данными (текст), либо пустыми элементами. Элемент на каждом уровне может рассматриваться в качестве корня своего собственного дерева (здесь можно воспользоваться термином *поддерево)*. Диаграмма дерева из предыдущего примера приводится на рис. 2.1.

Помимо «древесной» аналогии можно рассмотреть генеалогическую структуру XML. В этой структуре содержимое элементов (данные и элементы) рассматривается в качестве потомков этих элементов. Элементы, содержащие другие элементы, называются предками. В рассмотренном ранее примере перечня каждый элемент <item> является потомком одного и того же предка, элемента <list>, а также близнецом других элементов. Термины, связанные с деревьями и семейными взаимоотношениями, будут использоваться для дальнейшего описания взаимосвязей между элементами.

---

[1] В настоящее время появился стандартизованный XML-вариант HTML под названием XHTML. В связи с этим многие разработчики, имеющие дело с HTML, намерены включить поддержку грядущих XML-инструментов. При работе с XML различные типы разметки могут обрабатываться одними и теми же программами (редакторы, корректоры синтаксиса либо форматтеры). В скором времени разработка web-сайтов на основе HTML будет производиться с. помощью таких XML-языков, как DocBook и MathML.

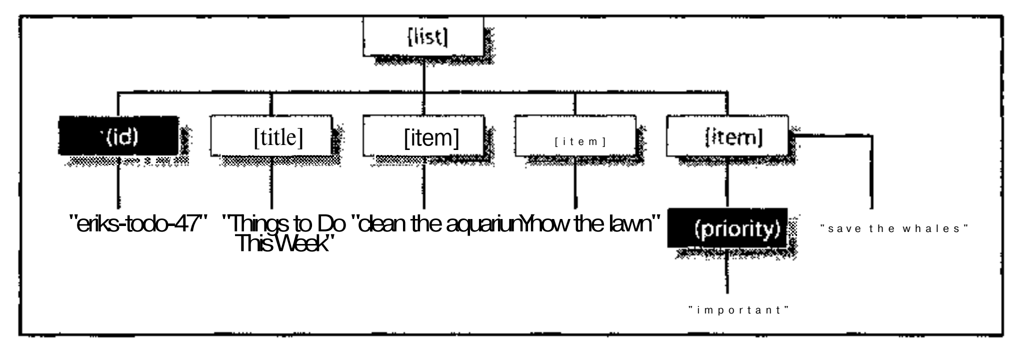


Рис. 2.1. Перечень задач, представленный в виде структуры дерева

Пространства имен

Элементы и атрибуты могут распределяться по группам, именуемым *пространствами имен.* Элементы и пространства имен связаны между собой примерно так же, как фамилия связана с конкретным человеком. Например, у троих ваших знакомых может быть имя Петр, но их фамилии будут различными. Практическое приложение этой концепции можно найти в листинге 2.2.

Листинг 2.2. Пример документа, включающего пространства имен

```
<?xml version="1.0"?>
<report>
    <title>Fish and Bicycles: A Connection?</title>
    <para>I have found a surprising relationship
    of fish to bicycles, expressed by the equation
    <equation>f = kb+n</equation>. The graph below illustrates
    the data curve of my experiment:</para>
    <chart xmlns:graph="http://mathstuff.com/dtds/chartml/">
        <graph:dimension>
            <graph:axis>fish</graph:axis>
            <graph:start>80</graph:start>
            <graph:end>99</graph:end>
            <graph:interval>1</graph:interval>
        </graph:dimension>
        <graph:dimension>
            <graph:axis>bicycle</graph:axis>
            <graph:start>0</graph:start>
            <graph:end>1000</graph:end>
            <graph:interval>50</graph:interval>
        </graph:dimension>
        <graph:equation>fish=0.01*bicycle+81.4</graph:equation>
    </graph:chart>
</report>
```

В примере рассматриваются два пространства имен. Первое — задано по умолчанию, причем в именах его элементов и атрибутов отсутствует двоеточие. Элементы, в именах которых содержится часть graph:, относятся к пространству имен «chartml» (вымышленное пространство имен, в области которого рисуется график). Часть элементов graph: образует *префикс пространства имен.* В результате включения префикса создается *уточняющее имя.* Элементы <equation> полностью от-

личаются друг от друга, поскольку играют в документе различные роли. Первый из элементов, находящийся в заданном по умолчанию пространстве имен, используется для форматирования математического уравнения. Второй элемент, относящийся к пространству имен графика, используется графической программой для рисования кривых линий.

Пространство имен всегда определяется элементом, включающим область применения данного пространства. Сам процесс определения выполняется с помощью атрибута в формате xmlns:*префикс*=*URL*. Здесь в качестве параметра *префикс* может использоваться префикс пространства имен (в рассматриваемом случае — `graph:`). Параметр *URL* представляет собой уникальный идентификатор, имеющий формат URL-ссылки либо другого идентификатора ресурса. Пространство имен не распознается за пределами области действия элемента.

Помимо поддержки двух типов элементов с одинаковыми именами либо типов атрибутов, пространства имен выполняют другие важные функции. Одна из функций заключается в облегчении форматирования документов средствами XML-процессора. Так, иногда изменения в пространстве имен могут свидетельствовать о том, что программа-форматтер, заданная по умолчанию, будет заменена приложением, предназначенным для форматирования специфических данных (например, графиков). В других случаях пространство имен инициирует толкование инструкций разметки в качестве метаразметки (в случае с XSLT).

Несмотря на очевидную пользу, пространства имен могут вызывать определенные проблемы. Позднее будет продемонстрировано, что пространства имен могут содержать объявления, ограничивающие применяемые типы элементов конечным набором. При использовании объявлений DTD применение пространств имен вызывает затруднения. Это связано с отсутствием специального средства, предназначенного для определения элементов и атрибутов, находящихся в пространстве имен (помимо тех, которые заданы по умолчанию). Причем эти объекты определяются в соответствии с неким другим XML-приложением. Частичная причина подобных затруднений кроется в том, что пространства имен появились в XML значительно позднее, чем формат объявлений DTD (возникновение последнего датируется эпохой господства SGML). Поэтому иногда пространства имен могут быть несовместимыми с некоторыми объявлениями DTD. На сегодняшний день эта проблема остается нерешенной, хотя и предпринимаются определенные попытки в этом направлении.

В главе 10 рассматриваются некоторые практические вопросы, возникающие при работе с пространствами имен.

Интервалы

Вы, наверное, уже обратили внимание па то, что во всех примерах книги используются элементы отступа и пробелы, улучшающие восприятие текста читателями. Подобный прием является весьма разумным, если постоянно приходится редактировать либо проверять XML-код. Однако в некоторых случаях применение интервалов может способствовать появлению нежелательных пробелов при печати. Поскольку правилами XML-языка не предусмотрено формулирование гипотез, связанных с данными, лишние пробелы автоматически не устраняются.

Существует решение, позволяющее сделать XML-процессор «умнее». Некоторые анализаторы могут определять, следует ли пропускать пробелы, в ходе выполнения обработки кода[1]. На основе объявления DTD они могут определять пробелы, добавленные с целью улучшения восприятия кода при чтении, и не являющиеся частью содержимого. Также можно настроить процессор на работу с определенным языком разметки. В результате процесса обучения процессор будет различным образом трактовать некоторые элементы, имеющие отношение к интервалам.

Если ни одна из указанных возможностей не решает проблему, возможно применение инструкций для XML-процессора, предотвращающих удаление пробелов. Зарезервированный атрибут xml:space может применяться любым элементом. Этот атрибут определяет, следует ли удалять пробел либо оставить его неизменным[2]. Ниже приводится соответствующий пример:

```
<address-label xml:space='preserve'>246 Marshmellow Ave.
Slumberville, MA
02149</address-label>
```

Здесь символы, определяющие перенос строки адреса, сохранены с целью дальнейшей обработки. Другим значением атрибута xml:space может быть "default". В этом случае XML-процессору предоставляется право выбора дальнейших действий, предпринимаемых по отношению к дополнительным пробелам.

Сущности

Дополнительные удобства для разработчиков XML-документов обеспечиваются с помощью другого свойства, именуемого *сущностью.* Это свойство может оказаться полезным в случае, если требуется контейнер для текста либо разметки, обычный ввод которых в силу различных причин затруднен. При этом часть XML-кода размещается за пределами документа[3], для связи фрагмента кода со всем документом используется *объектная ссылка.* XML-процессор должен подставлять вместо объектных ссылок текст на этапе синтаксического разбора. Поэтому каждая сущность, на которую указывает ссылка, должна быть объявлена. В результате процессор получает возможность разрешения ссылок.

Сущности объявляются в *объявлении типа документа (Document Type Declaration, DTD).* Этот объект состоит из двух частей. *Внутреннее подмножество* является составной частью документа, а *внешнее подмножество* находится в другом документе. (Довольно часто внешнее подмножество называется «DTD», а внутреннее подмножество сохраняет свое имя, несмотря на то, что оба подмножества образуют объявление DTD в целом.) В любом случае метод, применяемый для объявления сущностей, будет тем же самым. Практическое применение этого свойства демонстрируется в листинге 2.3.

---

[1] Анализатор является специализированным XML-обработчиком, выполняющим предварительный разбор документа. Различным анализаторам присуши разные уровни «интеллекта», проявляющегося в процессе интерпретации XML-кода. Этот вопрос более подробно будет рассматриваться в главе 3.

[2] Префикс «xml» говорит о наличии признака резервирования. Значение и области применения элементов и атрибутов с подобным префиксом определяются в XML-стандарте.

[3] На самом деле весь документ можно рассматривать в качестве одной сущности, именуемой *сущностью документа.* Однако термин «сущность» обычно применяется для обозначения части документа.

Листинг 2.3. Документ, содержащий определения сущностей

```
<!DOCTYPE memo
    SYSTEM "/xml-dtds/memo.dtd"
[
    <!ENTITY companyname "Willy Wonka's Chocolate Factory">
    <!ENTITY healthplan SYSTEM "hp.txt">
]>
<memo>
    <to>All Oompa-loompas</to>
    <para>
        &companyname; has a new owner and CEO, Charlie Bucket. Since
        our name, &companyname;, has considerable brand recognition,
        the board has decided not to change it. However, at Charlie's
        request, we will be changing our healthcare provider to the
        more comprehensive &Uuml;mpacare, which has better facilities
        for 'Loompas (text of the plan to follow). Thank you for working
        at &companyname;!
    </para>
    &healthplan;
</memo>
```

Рассмотрим данный пример подробнее. В первых строках содержится объявление DTD, представляющее собой специальную инструкцию разметки, которая включает большое количество важных данных, а так же определяет внутреннее подмножество и путь к внешнему подмножеству. Как и в случае любой другой декларативной разметки (при определении чего-либо нового), код начинается знаком восклицания, за которым следует ключевое слово, DOCTYPE. После ключевого слова указывается имя элемента, который будет использован для включения документа. Этот элемент называется *корневым элементом* либо *элементом документа.* Затем указывается путь к внешнему подмножеству, SYSTEM "/xml-dtds/memo.dtd", а также объявления внутреннего подмножества, заключенные в квадратные скобки ([ ]).

Во внешнем подмножестве находятся объявления, которые могут применяться во многих документах. Естественно, что это множество находится в другом файле. Внутреннее подмножество идеально для размещения объявлений, являющихся локальными для данного документа. Они могут отменять объявления, содержащиеся во внешнем подмножестве, либо расширять их набор. В примере две сущности объявлены во внутреннем подмножестве. Объявление сущностей содержит два параметра: имя сущности и текст замены. В данном случае сущности называются companyname и healthplan.

Упомянутые объекты имеют отношение к общим сущностям. Они отличаются от других видов сущностей, поскольку объявлены самим разработчиком. Текст замены, применяемый в этом случае, может находиться в двух разных местах. Первое объявление сущности определяет текст внутри самого объявления. Второе объявление указывает другой файл, в котором находится текст. Здесь применяется системный идентификатор, указывающий местоположение файлов. Этот идентификатор напоминает гиперссылку URL, применяемую web-браузером при поиске загружаемых страниц. В этом случае файл включается XML-процессором в исходном виде в любом месте, на которое ссылается так называемая *внешняя сущность.*

В ходе внимательного изучения примера нетрудно заметить инструкции разметки, выраженные в виде *&name;*. Знак амперсанда (&) свидетельствует о наличии объектной ссылки, параметр *name* определяет имя сущности, для которой ус-

танавливается ссылка. Одна и та же ссылка может использоваться повторно, благодаря чему обеспечивается удобный метод включения повторяющетося текста либо разметки (в качестве примера можно рассматривать сущность `companyname`).

Сущность может включать разметку (а также текст), как это было в случае с `healthplan` (фактически мы не знаем, что включается, так как сущность находится в другом файле, но поскольку она может выступать в качестве большого документа, то может включать как текст, так и разметку). Кроме того, могут включаться другие сущности, причем допускается произвольный уровень вложенности. Единственное ограничение заключается в том, что сущности не могут содержать сами себя. Благодаря этому предотвращается замыкающее определение, которое не может быть сконструировано с помощью XML-процессора. Использование рекурсивной логики допускается некоторыми XML-технологиями, такими как XSLT, по не стоит применять объектные ссылки для формирования кода, включающего круговые ссылки. Подобный код неприемлем для многих анализаторов.

И наконец, объявляется объектная ссылка, `Ü` Эта ссылка включается в любом месте внешнего подмножества и служит для отображения символов, которые не могут быть визуализированы устаревшими текстовыми редакторами. В данном случае требуется отображать заглавную литеру 'U' со знаком умляута: 0. Поскольку сущность, для которой устанавливается ссылка, состоит из одного символа, в роли ссылки будет применяться псевдоним, а не указатель. Обычный способ, применяемый для обработки «экзотических» символов (встроенный в XML-спецификацию), предусматривает использование числовой символьной сущности. В этом случае она выражается в виде символов `&#00DC`. Набор символов `0x00DC` представляет собой шестнадцатеричный эквивалент числа 220, определяющего положение символа U в таблице Unicode (набор символов, применяемый в XML, который рассматривается в следующей главе).

Поскольку сокращенное описательное имя, такое как `Uuml`, в общем случае проще для запоминания, чем набор символов 00DC, многие программисты на XML размещают подобные типы псевдонимов в объявлениях DTD для своих документов:

```
<!ENTITY % Uuml &#x00DC;>
```

В XML распознаются лишь пять встроенных именованных объектных ссылок, показанных в табл. 2.1. На самом деле эти объекты не являются ссылками в полном понимании этого слова, а представляют собой наборы символов, имеющих специальное значение для XML.

Таблица 2.1 Объектные ссылки в XML

Символ	Сущность
<	`<`
>	`>`
&	`&`
"	`"`
'	`'`

Лишь только две объектных ссылки применяются повсеместно в любом XML-документе: `&lt` и `&amp ;`. Теги элементов и объектные ссылки могут отображаться в любом месте документа. Например, никакой анализатор не может предположить, какую функцию выполняет в данный момент символ <, — функцию математиче-

ского символа «меньше» либо роль XML-маркера. Обычно анализатор выбирает последний вариант и сообщает о некорректно сформатированном документе в случае, если на самом деле мы имеем дело со знаком «меньше».

Наборы символов, кодировки и Unicode

Компьютеры воспринимают текст в качестве набора положительных целых чисел, соответствующих наборам символов. В свою очередь, наборы символов — это коллекции нумерованных символов (также называемых управляющими кодами), образующие некоторые стандартные конфигурации. Наиболее часто применяется «почтенный» набор символов US-ASCII. Этот набор состоит из 128 символов, включая прописные и строчные буквы латинского алфавита, числа, различные символы, символы пробелов различных типов, а также специальные коды печати, унаследованные со времен терминальных телетайпов. Подобная 7-битовая система может быть легко расширена путем добавления 8-го бита. В результате количество возможных символов удваивается. Примером подобного расширенного набора символов может служить ISO-Latin1, который применяется во многих Unix-системах. В набор ISO-Latin1 включены другие символы европейских языков, например, латинские буквы со знаками ударения, символы исландского языка, лигатуры, маркеры сносок, а также символы, применяемые при оформлении юридических документов. Помимо этого существует множество других лингвистических символов, которые могут быть «втиснуты» в 8-битовую систему.

Исходя из этих соображений, разрабатывалась новая архитектура кодирования символов, получившая название Unicode. Эта архитектура получила признание в качестве стандартного метода представления сценариев, предназначенных для хранения данных. В зависимости от реализации, набор символов Unicode может использовать до 32 битов для описания различных символов. В результате обеспечивается возможность для кодировки миллионов различных символов. Буквально за десять лет деятельности организации Unicode Consortium это пространство заполнялось наборами символами, включающими иероглифы китайского языка, различные математические, пунктуационные и знаковые символы. Несмотря на это все еще остается достаточно свободного места для включения новых символов на протяжении, как минимум, двух ближайших тысячелетий.

Вы, наверное, вовсе не удивитесь, узнав, что XML-документ может использовать любой тип кодировки. По умолчанию предполагается применение кодировки в стиле Unicode, переменной длины, известной под названием UTF-8. Данная кодировка предполагает применение от одного до шести байтов, применяемых для кодирования чисел, которые представляют адреса символов Unicode, а также длину символов, выраженную в байтах (в случае, если величина адреса превышает 255). Конечно, можно создать весь документ с применением 1-байтовых символов, причем в этом случае вы недалеко уйдете от ISO Latin-1 (небольшой массив адресов, причем сами адреса варьируются в диапазоне от 0 до 255). Однако в этом случае могут возникнуть проблемы, когда, например, вам понадобится символ, выходящий за пределы данного диапазона (скажем, при необходимости организации хранения символов одного из языков народов Юго-Восточной Азии). Здесь лучше воспользоваться кодировкой UTF-8. Процессоры, способные обрабатывать набор символов Unicode, выполняют подобную задачу корректно с последующим отображением правильных

символов. Устаревшие приложения просто игнорируют многобайтные символы. Начиная с версии 5.6, Perl может корректно обрабатывать наборы символов UTF-8. Эти вопросы подробнее будут рассматриваться в главе 3.

XML-объявления

После прочтения предыдущего раздела у проницательного читателя может возникнуть вопрос о целесообразности объявления метода кодирования в документе, если XML-процессор распознает кодировку в автоматическом режиме. В качестве ответа на этот вопрос можно сказать лишь то, что этот метод следует указывать в *XML-объявлении,* которое представляет собой строку текста, находящуюся в заголовке документа. Здесь описывается тип применяемой разметки, включая версию XML, кодировку символов, а также указывается, нуждается ли документ во внешнем подмножестве объявления DTD. Подобное объявление может выглядеть следующим образом:

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
```

Само объявление не является обязательным, как и каждый из его параметров (за исключением обязательного для указания атрибута `version`). Параметр, указывающий кодировку, требуется только в том случае, если используется кодировка символов, отличная от UTF-8 (поскольку данная кодировка определена по умолчанию). В случае, если указан параметр `"yes"` и если документ ссылается на внешние сущности, наличие автономного объявления приведет к тому, что в ходе проверки с помощью анализатора генерируется ошибка.

Инструкции по обработке и другие структурные элементы разметки

Помимо рассмотренных ранее элементов, XML предоставляет в распоряжение пользователя некоторые другие синтаксические объекты. *Инструкции по обработке* (`processing instructions`, PI) применяются для передачи информации выбранному XML-процессору. Требуемый процессор указывается с помощью параметра *цель,* за которым следует необязательный параметр *данные.* Если в программе не распознается параметр *цель,* инструкция PI пропускается и создается впечатление, что она вообще не существует. Ниже приводится пример, основанный на негласном хакерском опыте авторов книги:

```
<?file-breaker start chap04.xml?><chapter>
<title>The very long title<?lb?>that seemed to go on forever and ever</title>
<?xml2pdf vspace 10pt?>
```

Цель первой инструкции PI — `file-breaker`, а соответствующие ей данные содержатся в файле chap04.xml. Программа, считывающая этот документ, ищет инструкцию с заданным целевым ключевым словом, а затем выполняет обработку данных. Реализуемая в данном случае цель заключается в создании нового файла и в сохранении в нем следующего XML-кода.

Вторая инструкция содержит только один целевой объект — `lb`. Этот пример фактически определяет передачу XML-процессору команды, определяющей

создание разрыва строки в этом месте текста. При выполнении примера возникают две проблемы. Инструкция Р1 выполняет замену символа пробела. Проблема в этом случае возникает тогда, когда программа не может распознать эту инструкцию. В результате будет утеряно значение символа пробела, заключающееся в разделении двух слов. Лучше поместить пробел после инструкции PI и указать процессору на необходимость удаления всех последующих символов пробела. Вторая проблема возникает из-за применения в качестве цели инструкции, а не фактического имени программы. Поэтому лучше воспользоваться более характерным именем xml2pdf, указав его после инструкции PI. Используемое в данном случае имя лучше, поскольку lb слишком уж напоминает элемент данных.

Инструкции PI весьма удобны для разработчиков. Не существует общего правила, определяющего формат имени цели либо применяемых данных, но, как правило, имена являются более специфичными, а обозначения для данных — очень краткими.

Если вы имели опыт написания документов с применением средств встроенного языка миниразметки Perl (Plain Old Documentation, Простая старая документация)[1], то заметите подобие некоторых инструкций PI и POD-директив. В качестве примера можно рассматривать параграфы =for и блоки =begin/=end. В параграфах либо блоках можно оставлять для POD-процессора небольшие сообщения, в которых определена цель и некоторые аргументы (либо просто произвольная строка текста).

К другим часто используемым объектам разметки можно отнести *XML-комментарии.* По определению, комментарии представляют собой фрагменты текста, игнорируемые XML-процессором. Назначение комментариев заключается в сообщении какой-либо информации читателям (как правило, в виде различных примечаний). Они также полезны при временном «отключении» части кода разметки па этапе отладки документа. Эта процедура безопаснее, чем простое удаление фрагментов текста. Ниже приводится соответствующий пример:

```
<!-- Следующий фрагмент текста не воспринимается анализатором -->
This is perfectly visible XML content.
<!--
    <para> This paragraph is no longer part of the document.</para>
-->
```

Обратите внимание на сходство синтаксиса и порядка обработки для XML- и HTML-комментариев.

Вложенные комментарии не допускаются. Причем не допускается даже «бледная тень» вложенного комментария. Так, например, в комментариях запрещено использовать строки "--" независимо от их назначения.

А теперь приступим к рассмотрению *секции CDATA.* Эта конструкция предназначена для определения символьных данных, которые не могут быть разобраны с помощью XML-анализатора. Другими словами, секция CDATA не содержит элементов языка разметки. Это полезно в случае, если требуется включить блок текста, состоящий из недопустимых символов языка (например, <, > и &), а преобразование в символьные объектные ссылки может быть затруднено.

---

[1] *Водолазкий В.* Энциклопедия Perl. — СПб.: Питер, 2001.

Ниже приводится соответствующий пример:

```
<codelisting>
<![CDATA[if( $val > 3 && @lines ) {
   $input = <FILE>;
}]]>
</codelisting>
```

Содержимое, заключенное между символами `<![CDATA[` и `]]>`, трактуется в качестве данных, которые не имеют ничего общего с языком разметки. Секции CDATA в дальнейшем будут применяться относительно редко, поскольку они «захламляют» написанный текст, а также затрудняют обработку XML-кода. Впрочем, бывают ситуации, когда без этих конструкций просто не обойтись[1].

XML-документы: свободно определенная форма и формальная корректность

Синтаксис языка SGML требовал полного объявления каждого элемента и атрибута в длинном перечне объявлений DTD. Описание порядка объявления находится в следующем параграфе, а теперь просто представьте себе, что вы имеете дело с калькой документа. При использовании подобной кальки обработка документа будет связана с немалыми накладными расходами, вследствие чего серьезно страдает реноме SGML как языка разметки, используемого в Интернете. Язык HTML изначально позиционировался в качестве подмножества SGML, вследствие чего он перенял «лучшие» качества своего предшественника. Поэтому любой «правильный» HTML-документ должен соответствовать объявлению HTML DTD. В любом случае расширение возможностей языка невозможно без получения одобрения web-комитета.

В XML подобное требование обходится путем добавления специального условия, именуемого *свободно определенной формой XML*. В этом режиме при составлении документа требуется соблюдение минимального набора синтаксических правил. Если эти правила соблюдаются, документ считается *формально корректным*. Эти правила обеспечивают невиданную степень свободы для разработчиков, поскольку уже не требуется просмотр объявлений DTD каждый раз, когда необходимо обработать часть XML-кода. В обязанности процессора входит проверка соблюдения минимальных синтаксических правил.

В XML-документе, имеющем свободно определенную форму, можно указывать любые имена элементов. Они не определяются в предварительно сформированном словаре, как в случае с HTML-кодом. Придание некоторой степени свободы языку разметки, используемому в программах, сопряжено с определенным риском, по если вы контролируете ситуацию, ничего страшного не произойдет. Если же вы полагаете, что разметка не будет соответствовать желаемому шаблону, воспользуйтесь объявлениями элементов и атрибутов (см. следующий раздел).

---

[1] Авторы книги воспользовались секциями CDATA при оформлении XML-подобного документа DocBook, с помощью которого осуществлялась подготовка книги к печати. В частности, в эти секции были помещены все листинги кода и примеры XML-документов, благодаря чему исключалась обработка недопустимых символов, < и &.

Каковы же правила формальной корректности? Ниже представлен краткий перечень этих правил:

- документ может содержать только один элемент верхнего уровня, именуемый *элементом документа*, который, в свою очередь, содержит другие элементы и данные. Все XML-объявления и определения типа документа должны предшествовать элементу документа;
- каждый элемент содержимого документа должен включать начальный и конечный тег;
- имена элементов и атрибутов чувствительны к смене регистра символов, причем могут использоваться лишь определенные виды символов (буквы, символы подчеркивания, дефисы, точки и числа), а в качестве первого символа могут применяться только буквы и символы подчеркивания. Допускаются двоеточия, но только в префиксе объявленного пространства имен;
- все атрибуты должны иметь значения, заключенные в кавычки.
- не допускается перекрытие элементов; начальный и конечный теги элемента должны четко выделяться внутри одного и того же элемента;
- некоторыеэлементы, включая угловые скобки (< >) и знакамперсанда (&), зарезервированы языком разметки и не допускаются в качестве части содержимого, подвергаемого синтаксическому разбору. Вместо них воспользуйтесь символьными объектными ссылками либо просто недопустимое содержимое поместите в секцию CDATA;
- синтаксис, применяемый для начальных тегов непустых элементов, отличается от синтаксиса пустых элементов. В последнем случае синтаксические правила требуют указание символа слэша (/) перед закрывающей угловой скобкой (>) тега.

Существуют и дополнительные правила, поэтому для получения более полного представления относительно принципов формальной корректности необходимо обратиться к дополнительной литературе либо прочесть официальные рекомендации консорциума W3C, помещенные на web-узле http://www.w3.org/XML.

Если необходимо выполнить обработку документа с помощью XML-приложений, убедитесь в том, что он соответствует требованиям формальной корректности. Инструмент, применяемый для проверки соблюдения соответствующих правил, называется *программой проверки формальной корректности*. В роли этой программы может быть использован XML-анализатор, сообщающий пользователю сведения о найденных ошибках. Часто подобный инструмент выполняет детальный анализ, в результате чего может сообщаться даже номер строки в файле, в которой произошла ошибка. Программы проверки и анализаторы подробно рассматриваются в главе 3.

Объявление элементов и атрибутов

Если требуется дополнительный уровень контроля качества (помимо сводки о «состоянии здоровья», формируемой на основе метки «формальной корректности»), в объявлении DTD потребуется определить грамматические шаблоны для языка разметки. Благодаря этому получаем формальный язык, документированный

согласно международным стандартам. При наличии объявления DTD программа получает средства для определения степени согласованности документа или, говоря другими словами, пример правильного оформления документа.

В DTD определяются два типа объявлений, предназначенных для установки модели языка. Первый тип называется *объявлением элемента.* В результате его применения в набор разрешенных элементов добавляется новое имя, а в специальном языковом шаблоне указываются допустимые составляющие компоненты для элемента. Ниже приводятся некоторые примеры:

```
<!ELEMENT sandwich ((meat | cheese)+ | (peanut-butter, jelly)),
condiment+, pickle?)>
<!ELEMENT pickle EMPTY>
<!ELEMENT condiment (PCDATA | mustard | ketchup )*>
```

Первый параметр объявляет имя элемента. Второй параметр представляет собой шаблон (модель содержимого, заключенная в скобки, либо такое ключевое слово, как EMPTY). Синтаксис моделей содержимого напоминает синтаксис регулярных выражений. В случае с моделями основное различие заключается в том, что имена элементов представляют собой символы, причем запятая применяется с целью указания требуемой последовательности элементов. Каждый элемент, упомянутый в модели содержимого, должен объявляться в DTD.

Другой важный тип объявлений именуется *объявлением списка атрибутов.* Этот список позволяет объявлять набор дополнительных либо обязательных атрибутов для данного элемента. Значения атрибута должны контролироваться в некотором диапазоне, хотя и ограничения шаблона в чем-то ограничены. Обратите внимание на следующий пример:

```
<!ATTLIST sandwich
    id      ID       #REQUIRED
    price   CDATA    #IMPLIED
    taste   CDATA    #FIXED   "yummy"
    name    (reuben | ham-n-cheese | BLT | PB-n-J )   'BLT'
>
```

Общий шаблон объявления атрибутов состоит из трех частей: имя, тип данных и поведение. В примере объявляются три атрибута для элемента `<sandwich>`. Первый из них, id, представляет собой идентификатор типа. Он может иметь вид уникальной символьной строки, применяемой только один раз в составе любого атрибута идентификатора типа в произвольном разделе документа. Идентификатор обязателен для применения (ключевое слово #REQUIRED). Второй атрибут, price, представляет собой тип CDATA. Этот атрибут необязателен в соответствии с ключевым словом #IMPLIED. Третий атрибут, taste, представляет собой фиксированное значение "yummy", которое не может изменяться (все элементы <sandwich> наследуют этот атрибут автоматически). И наконец, атрибут name выбирается из нумерованного списка значений (по умолчанию, 'BLT').

Объявления элементов и атрибутов обладают некоторыми недостатками. Модели содержимого присуща не слишком высокая степень гибкости. Например, довольно трудно закодировать инструкцию «этот элемент должен содержать каждый из элементов A, B, C и D в произвольном порядке» (попробуйте и убедитесь в этом сами!). Кроме того, невозможно установить ограничения для символьных данных. Невозможно проверить, что тег <date> содержит корректную дату, а не, например, почтовый адрес. Причина третьего наибольшего затруднения заключает-

ся в самом сообществе программистов, применяющих язык XML. Причина проблем кроется в плохом сопряжении между объявлениями DTD и пространствами имен. При использовании объявлений элементов требуется определять все элементы, которые будут использоваться в документе, а не ограничиваться лишь некоторыми из них. Если же вы хотите оставить возможность для импорта элементов некоторых типов из других пространств имен, использование объявлений DTD с целью проверки документов становится весьма затруднительным. Проблему можно обойти, если воспользоваться смешанными DTD-комбинациямн, как было описано ранее, но этот метод не всегда приемлем.

Схемы

Недостатки, связанные с применением DTD-объявлений, мотут быть устранены с помощью альтернативных языковых схем. Консорциум W3C в подобных случаях рекомендует использовать язык XML Schema. Следует знать, что этот язык является лишь одним из многих конкурирующих языков, имеющих тип схемы. Вполне возможно, что вам подойдет другой язык из этого семейства. Если вы решились на выбор другой схемы, обратитесь к сети CPAN на предмет поиска модуля, призванного наилучшим образом выполнить ваши требования.

В отличие от объявлений DTD, схемы XML Schema являются XML-документами. Реальное преимущество, достигаемое при использовании схем, заключается в возможности детализованного контроля над формами, с помощью которых осуществляется ввод пользовательских данных. Наличие подобного контроля особенно приветствуется в документах, для которых проверка достоверности вводимых данных является столь же важной, как и наличие подходящей структуры. Листинг 2.4 демонстрирует схему, разработанную для форм сбора сведений о моделях. В этих формах обязательной является проверка типа данных.

Лиситнг 2.4. XML-схема

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema-instance">

    <xs:annotation>
        <xs:documentation>
        Census form for the Republic of Oz
        Department of Paperwork, Emerald City
    </xs:documentation>
</xs:annotation>

<xs:element name="census" type="CensusType"/>

<xs:complexType name="CensusType">
    <xs:element name="censustaker" type="xs:decimal" minoccurs="0"/>
    <xs:element name="address" type="Address"/>
    <xs:element name="occupants" type="Occupants"/>
    <xs:attribute name="date" type="xs:date"/>
</xs:complexType>

<xs:complexType name="Address">
    <xs:element name="number" type="xs:decimal"/>
    <xs:element name="street" type="xs:string"/>
    <xs:element name="city"   type="xs:string"/>
    <xs:element name="province"  type="xs:string"/>
    <xs:attribute name="postalcode" type="PCode"/>
</xs:complexType>
```

```
<xs:simpleType name="PCode" base="xs:string">
    <xs:patternvalue = "[A-Z]-d{3}"/>
</xs:simpleType>

<xs:complexTypename = "Occupants">
    <xs:element name="occupant" minOccurs="1" maxOccurs="20">
        <xs:complexType>
            <xs:element name="firstname" type="xs:string"/>
            <xs:element name="surname" type="xs:string"/>
            <xs:element name="age">
                <xs:simpleType base="xs:positive-integer">
                    <xs:maxExclusive value="200"/>
                </xs:simpleType>
            </xs:element>
        </xs:complexType>
    </xs:element>
</xs:complexType>

</xs:schema>
```

В первой строке кода производится идентификация документа в качестве схемы, а также выполняется его связывание с пространством имен XML Schema. Следующая структура, <annotation>, в основном предназначена для описания назначения документа схемы. После документальной части начинается объявление типов элементов.

Мы начнем с объявления корня нашего типа документа — с элемента, называемого <census>. Само объявление представляет собой элемент, имеющий тип <xs:element>. Соответствующие атрибуты элемента получают имя "census", а тип описания для <census > будет «CensusType». В схемах, в отличие от DTD, описания содержимого часто хранятся отдельно от объявлений, благодаря чему облегчается определение обобщенных типов элементов, а также назначение им различных элементов. При дальнейшей детализации схемы обнаруживается описание фактического содержимого— элемент <xs :complexType> с именем name="CensusType". Здесь определяется, что элемент <census> содержит необязательный элемент <censustaker>, за которым следуют обязательные элементы <occupants> и <address>. Также должен определяться атрибут date.

Атрибут date и элемент <censustaker> обладают специфическими шаблонами данных, назначенными в описании для <census>: дата и десятичное число. Если в качестве содержимого документа <census> используется числовое значение, это, согласно применяемой схеме, может привести к ошибке. Подобный уровень контроля не может быть получен с помощью объявления DTD.

Многие типы схем поддаются проверке. К ним можно отнести числовые значения (байты), числа с плавающей точкой, длинные целые числа, двоичные числа, а также булевы значения; шаблоны для маркировки времени и длительности; адреса Интернета, а также URL-ссылки; идентификаторы, ссылки идентификаторов и другие типы, позаимствованные из объявления DTD; символьные строки.

В описании тина элемента используются свойства, называемые *аспектами (facet),* либо даже более детализованные границы содержимого. Например, в приведенном выше примере, схема включает элемент <age>, тип данных которого — положительное целое число, а максимальное значение, равное 200, применяется для аспекта max-inclusive. Схема XML Schema включает многие другие аспекты, в том числе precision, scale, encoding, pattern, enumeration и max-length.

Описание Address определяет новую концепцию: шаблоны, определенные пользователем. Благодаря применению этой методики, postalcode может определяться с помощью шаблонного кода: [A-Z] -d{3}. Использование этого кода подобно высказыванию: «Приемлемы любые алфавитные символы, за которыми следует дефис и три цифры». Если вас не устраивает ни один из применяемых типов данных, всегда можно разработать свой собственный тип.

Схемы представляют собой пример новой технологии, облегчающей применение XML, особенно в случае работы с приложениями, обрабатывающими данные (формы ввода данных). Ограниченный объем книги не позволит углубляться в детали, поэтому за дополнительными сведениями обращайтесь к другим источникам.

Другие стратегии работы со схемами

С момента получения одобрения консорциума W3C, инструмент XML Schema эволюционировал и ныне представляет собой не просто схему, позволяющую выполнять гибкую проверку достоверности документа. Некоторые программисты предпочитают использовать методы, доступные посредством внедрения таких спецификаций, как RelaxNG (доступна па web-узле http://www.oasis-open.org/committees/relax-ng/) либо Schematron (http://www.ascc.net/xml/resource/schematron/schematron.html). При этом те же самые цели могут достигаться с помощью различных философских подходов. Последняя спецификация включала реализации языка Perl, доступные в настоящее время. Более подробно эта спецификация будет рассмотрена в главе 3.

Трансформации

А сейчас приступим к рассмотрению последнего вопроса этой главы — концепции трансформаций. В XML *трансформация* определяется как процесс реструктуризации либо преобразования документа в другую форму. При выполнении трансформаций консорциум W3C рекомендует воспользоваться языком трансформирования XML, который называется языком таблиц стилей, применяемых для выполнения XML-трансформации (XSLT, XML Stylesheet Language for Transformations). Эта технология является весьма полезной и удобной в работе.

Подобно языку XML Schema, XSLT-сценарий трансформации представляет собой XML-документ. Его создание регулируется *правилами шаблона,* каждое из которых является инструкцией, определяющей преобразование типов элементов. Термин *шаблон* часто применяется в смысле определения внешнего вида того или иного объекта. При этом подразумеваются пустые места, заполняемые какими-либо данными в дальнейшем. А теперь обратите внимание на точное определение правил шаблона: они определяют внешний вид конечного документа, а зарезервированные места заполняются данными XSLT-процессора.

В листинге 2.5 приводится упрощенная информация, с помощью которой простой XML-документ, DocBook, преобразуется в HTML-страницу.

Листинг 2.5. Документ, представляющий пример XSLT-трансформации

```
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">
```

```
<xsl:output method="html"/>

<!-- ПРАВИЛО ДЛЯ ЭЛЕМЕНТА BOOK -->
<xsl:template match="book">
    <html>
        <head>
            <title><xsl:value-of select="title"/></title>
        </head>
        <body>
            <h1><xsl:value-of select="title"/></h1>
            <h3>Table of Contents</h3>
            <xsl:call-template name="toc"/>
            <xsl:apply-templates select=" chapter"/>
        </body>
    </html>
</xsl:template>

<!-- ПРАВИЛО ДЛЯ CHAPTER -->
<xsl:template match="chapter">
    <xsl:apply-templates/>
</xsl:template>

<!-- ПРАВИЛО ДЛЯ CHAPTER TITLE -->
<xsl:template match="chapter/title">
    <h2>
        <xsl:text>Chapter </xsl:text>
        <xsl:number count="chapter" level="any" format="1"/>
    </h2>
    <xsl:apply-templates/>
</xsl:template>

<!-- ПРАВИЛО ДЛЯ PARA -->
<xsl:template match="para">
    <p><xsl:apply-templates/></p>
</xsl:template>

<!-- ИМЕНОВАННОЕ ПРАВИЛО: TOC -->
<xsl:template name="toc">
    <xsl:if test="count(chapter)>0">
        <xsl:for-each select="chapter">
            <xsl:text>Chapter </xsl:text>
            <xsl:value-of select="position( )"/>
            <xsl:text>: </xsl:text>
            <i><xsl:value-of select="title"/></i>
            <br/>
        </xsl:for-each>
    </xsl:if>
</xsl:template>

</xsl: stylesheet>
```

Сначала XSLT-процессор считывает таблицу стилей и создает таблицу правил шаблона. Затем происходит синтаксический разбор XML-документа (предназначенного для преобразования) и выполняется поочередный обход узлов документа. *Узел* представляет собой элемент, в качестве которого может использоваться фрагмент текста, инструкция по обработке, атрибут либо объявление пространства имен. Для каждого узла XSLT-процессор пытается найти лучшее правило установки соответствия. Процессор применяет правило, выводит данные, поступающие от шаблона, и в случае необходимости переходит к выполнению других правил.

В листинге 2.6 содержится документ, подвергаемый трансформации.

Листинг 2.6. Трансформируемый документ

```
<book>
    <title>The Blathering Brains</title>
    <chapter>
        <title>At the Bazaar</title>
        <para> What a fantastic day it was. The crates were stacked
               high with imported goods: dates, bananas, dried meats,
               fine silks, and more things than I could imagine. As I
               walked around, savoring the fragrances of cinnamon and
               cardamom, I almost didn't notice a small booth with a
               little man selling brains.</para>
        <para>Brains! Yes, human brains, still quite moist and squishy,
               swimming in big glass jars full of some greenish
               fluid.</para>
        <para>"Would you like a brain, sir?" he asked. "Very
               reasonable prices. Here is Enrico Fermi's brain for
               only two dracmas. Or, perhaps, you would prefer
               Or the great emperor Akhnaten?"</para>
        <para>I recoiled in horror...</para>
    </chapter>
</book>
```

А теперь вкратце остановимся на описании процесса трансформации.

1. Первый элемент называется <book>. В качестве наилучшего правила установки соответствия можно воспользоваться первым элементом, поскольку он явно соответствует слову «book». В качестве тегов вывода шаблоном рассматриваются теги <html>, **<head> и** <title>. Обратите внимание, что эти теги трактуются как данные разметки, поскольку они не содержат префикс xsl: namespace.

2. После того как процессор получает XSLT-инструкцию <xsl:value-of select="title"/>, осуществляется поиск элемента <title>, который является вложенным для текущего элемента, <book>. Затем требуется получить значение этого элемента, в качестве которого используется содержащийся в нем текст. Этот текст выводится внутри элемента <title> в качестве непосредственного шаблона.

3. Затем процессор продолжает выполнять обработку, пока не встретит инструкцию <xsl:call-template name="toc"/>. Если вы посмотрите на нижнюю часть таблицы стилей, то увидите там правило шаблона, которое начинается строкой <xsl:template name="toc">. Это правило шаблона называется *именованным шаблоном* и по своему действию напоминает вызов функции. С его помощью собирается содержание и возвращается текст вызывающему правилу с целью дальнейшего вывода.

4. Внутри именованного шаблона находится элемент <xsl:if test="count (chapter) >0">. Этот элемент является условным оператором, проверяемое условие выполняет оценку количества элементов <chapter>, находящихся в текущем элементе (<book>). После прохождения проверки выполняется дальнейшая обработка внутри элемента.

5. Выполнение инструкции <xsl:for-each select="chapter"> приводит к тому, что процессор «посещает» каждый вложенный элемент <chapter> и времен-

но делает его текущим элементом внутри элемента <xsl:for-each>. Этот шаг аналогичен итерации цикла foreach() в Perl. Оператор <xsl:value-of select="position()"/> определяет числовую позицию для каждого элемента <chapter>, а затем выводит ее. В результате документ считывает строки "Chapter 1", "Chapter 2" и т. д.

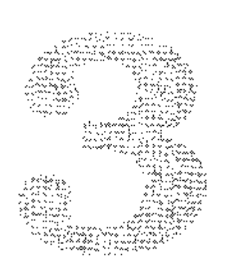
6. Именованный шаблон "toc" возвращает текст вызывающему правилу шаблона и продолжает выполнение сценария. Затем процессор получает директиву <xsl:apply-templates select="chapter"/>. Вывод <xsl:apply-templates> без указания каких-либо атрибутов означает, что процессор должен обрабатывать каждый из вложенных элементов для текущего элемента, делая их текущими. При наличии атрибута select="chapter" могут обрабатываться только те вложенные элементы, которые имеют тип <chapter>. После завершения обработки всех вложенных элементов и возврата инструкцией присущего ей текста, происходит отображение этого текста и переход к выполнению следующих правил.

7. При перемещении к первому элементу <chapter> процессор выбирает подходящее правило среди <xsl:apply-tempaltes/>. Оставшаяся часть обработки достаточно проста и сводится к выполнению правил для оставшихся элементов, <title> и <para>.

Язык XSLT обладает богатыми возможностями по выполнению трансформаций, но ему также присущи и некоторые недостатки. В частности, может замедляться обработка больших документов. Это связано с тем, что перед выполнением обработки создается внутреннее представление всего документа. Синтаксис данного языка обладает не столь богатыми выразительными способностями и простотой применения, как синтаксис языка Perl. В дальнейшем будут представлены некоторых проблем, которые могут решаться в рамках языка XSLT средствами Perl. Ну а вы сами сможете выбрать, что предпочесть, — простоту языка XSLT или мощь Perl.

Вот и завершился наш краткий тур по стране под названием XML. В следующей главе мы начнем рассмотрение основ обработки кода XML средствами Perl с применением анализаторов и базовых генераторов. Необходимо четко осознавать область применения XML, знакомиться с методами его применения, а также распознавать все конструкции, рассмотренные в этой главе. Если вы не уверены в своих силах, отложите в сторону эту книгу и освежите в памяти основы работы с XML.

Основы XML: чтение и запись



В настоящей главе освещаются две наиболее важные задачи, возникающие при работе с XML-документами: чтение и запись данных, находящихся в памяти. Известно, что XM.L определяет структурированный, предсказуемый и стандартный формат, предназначенный для храпения данных. Известно, что в Perl принят стиль обработки текста «построчно, выполнение-по-мере-продвижения». В XML предполагается, что пользователь уже ознакомлен с «правилами игры» (структуры и протоколы, описанные в главе 2), прежде чем приступает к работе. К счастью, большая часть тяжкого труда, связанного с обработкой текста, уже проделана создателями языка XML. Результаты этих усилий воплощены в виде модулей-анализаторов и ряда других инструментов. С некоторыми из них вы уже ознакомились в главе 1.

Знание принципов работы с синтаксическими анализаторами очень важно. Как правило, с их помощью выполняется обработка практически всех пользовательских данных. По крайней мере, анализаторы преобразуют данные таким образом, что становится возможной их обработка. Любой программист знает, что наличие подготовленных данных — это уже полдела. А теперь углубимся в дебри процесса синтаксического разбора и определимся с выбором требуемых стратегий.

Анализаторы имеют огромное количество опций, которые позволяют вам сконфигурировать вывод данных в соответствии с конкретными нуждами. Какого рода набор символов требуется в этой ситуации? Необходима ли проверка достоверности документа или достаточно ограничиться проверкой соблюдения правил формальной корректности? Нужно ли расширять объектные ссылки либо следует их оставить в исходном виде? Каким образом устанавливается обработчик управления событиями и как используется анализатор при создании дерева? Далее будут подробно рассмотрены эти возможности, в результате чего вы сможете более эффективно работать с синтаксическими анализаторами.

И напоследок будет продемонстрировано, как избежать некоторых проблем, связанных с неправильной кодировкой используемого XML-кода. Правильное выполнение этого шага довольно важно в случае, если требуется использовать ваши данные повторно без выполнения трудоемкой ручной правки.

XML-анализаторы

Операции ввод/вывода файлов свойственны любому языку программирования, но обычно они выполняются на низком уровне: считывание символов или строк, фильтрация ввода с применением регулярных выражений и т. д. Как правило, управление исходным текстом затруднено, а так же отсутствуют указания по разделению дискретных порций текста (если не учитывать используемую концепцию, заключающуюся в разделении строк символами новой строки и в разделении столбцов с помощью символов табуляции). Также существует множество схем упаковки данных, число которых превышает величины, возникающие в воображении летописцев Вавилона. И вот из всего этого многообразия выделяется XML, четкие правила которого обеспечивают такие возможности, как разграничение данных, распределение иерархий и выполнение ссылок на ресурсы предсказуемым образом. Программа, которая полагается на эти правила, может считывать XML-документ, отвечающий требованиям формальной корректности. В этом она напоминает персонаж, поместивший у себя в ухе «вавилонскую рыбу»[1].

Где же взять подобную рыбу, если в роли персонажа выступает ее величество Программа? По определению, *XML-анализатор* — это программа или библиотека кодов, выполняющая трансляцию XML-данных в поток событий либо в объект данных. В результате подобных действий пользовательская программа получает непосредственный доступ к структурированным данным. XML-код может выступать в виде символьного потока, статической строки, одного или нескольких файлов или дескрипторов файлов. Этот поток или строка могут быть «нафаршированы» объектными ссылками, которые требуется или не требуется разрешать. Некоторые данные могут поступать из внешнего мира (например, из отдаленных местоположений в Интернете). Кодировка символов может быть самой разнообразной (например, латиница либо японский язык). К счастью, программисту вряд ли потребуется учитывать какую-либо из этих деталей в своей программе, потому что обо всем позаботится анализатор. Программа анализатора представляет собой «абстрактный туннель» между физическим состоянием данных и их кристаллизованным представлением.

Программы XML-анализаторов связывают данные разметки (пакеты данных с внедренными XML-инструкциями) и некую исходную форму, с которой может работать ваша программа. В случае с Perl под исходной формой подразумевается совокупность хэш-объектов, скалярных элементов, массивов, а также объектов, появившихся в результате установки ссылок на другие объекты. XML-код может иметь сложную структуру, а его части помещаются во многие файлы или потоки. Также в его составе могут содержаться неразрешенные сущности, которые нуждаются в корректировке. Анализаторы же обычно принимают только качественный XML-код, не содержащий формальных ошибок. Вывод кода должен отражать структуру: порядок, содержание, ассоциативность данных, в то время как такие второстепенные детали, как источник данных и применяемый набор символов,

---

[1] Читатели книги Дугласа Адамса «Путеводитель по Галактике» (Douglas Adams *Hitchhiker's Guide to the Galaxy)* помнят, что вавилонская рыба — это живое существо, выполняющее функции переводчика с любых языков.

игнорируются. В результате возникает большой объем дополнительной работы. А теперь перечислим действия, выполняемые XML-анализатором:

- считывает поток символов и отделяет разметку от данных;
- по возможности заменяет объектные ссылки их значениями;
- собирает завершенный логически непротиворечивый документ на основе многих разрозненных источников;
- сообщает о синтаксических, а также (дополнительно) о грамматических ошибках;
- передает данные и структурную информацию клиентской программе.

В XML данные и разметка не разделены, поэтому анализатор должен отфильтровать поток символов и выполнить необходимые операции по разделению данных. Определенные символы отделяют команды от данных: для элементов, комментариев и инструкций по обработке применяются угловые скобки, а для объектных ссылок используются символы амперсанда и точка с запятой. Анализатор также может ожидать определенные команды и распознает неверные команды (например, в случае если имеется начальный, но отсутствует завершающий тег). Располагая подобной информацией, анализатор может преобразовать символьный поток в дискретные части, соответствующие XM L-разметке.

Следующая задача состоит в том, чтобы заполнить размеченные области. Для этого может понадобиться разрешить *объектные ссылки.* На первом этапе, в процессе считывания XML-кода процессор находит перечень определений меток-заполнителей в виде объявлений сущностей. Причем краткий идентификатор ассоциируется с объектом. Идентификатор — это некоторый буквенный текст, находящийся в определении DTD для данного документа. Сущность может также задаваться в этой схеме или в деловой части URL-ссылки. Эти объекты могут, в свою очередь, содержать другие объектные ссылки, так что процесс разрешения ссылок может быть многоэтапным и завершиться после того, как будут окончательно заполнены размеченные области.

Разрешение сущностей требуется далеко не всегда. Если вы просто отказываетесь от использования XML-документа после какой-нибудь незначительной обработки, то может потребоваться отключить разрешение сущности или воспользоваться собственной процедурой для обработки объектных ссылок. Также можно разрешать внешние объектные ссылки (для сущностей, чьи значения находятся во внешних местоположениях, на которые указывают URL-ссылки), но не разрешать внутренние ссылки. Большинство анализаторов предоставляют подобные возможности, но не разрешают использовать объектные ссылки без их объявления.

Изложенные выше соображения приводят к формулированию третьей задачи. Если позволить анализатору разрешать внешние сущности, будет осуществлена выборка всех документов, локальных и удаленных, которые содержат часть более обширного XML-документа. При осуществлении этого процесса все сущности будут объединены, образуя единый документ. Если программа не нуждается в информации о физической структуре документа, данные об источнике какой-либо части документа становятся излишними непосредственно после завершения объединения всего документа.

При осуществлении интерпретации разметки анализатор может обнаруживать синтаксические ошибки. В процессе проектирования XML-языка предусматривал-

ся простой алгоритм ошибок подобного рода. Для всего, начиная с атрибутов и заканчивая тегами пустого элемента, установлены жесткие правила, формирующие их внутреннее устройство. Например, следующий фрагмент XML-кода имеет очевидную ошибку. Начальный тег для элемента <decree> содержит атрибут, которому присвоено некорректное значение. В значении 'now" пропущен второй символ кавычки, а также имеется ошибка в завершающем теге. Постарайтесь ее обнаружить.

```
<decree effectives"now>All motorbikes
shall be painted red.</decree<
```

В случае возникновения подобных ошибок анализатор может прекратить выполняемую операцию. Нет никакого смысла в попытке анализа оставшейся части документа. Назначение XML заключается в соблюдении однозначности. Если анализатор должен угадывать, как может выглядеть документ[1], соответствующие данные становятся неопределенными, а надежность такой программы катастрофически падает. Вместо этого структура языка XML была спроектирована таким образом, чтобы XML-анализаторы не пропускали «плохие» XML-документы. Если анализатору «нравится» ваш XML-код, он выдает сообщение, что код *формально корректен.*

Что же мы подразумеваем под «грамматическими ошибками»? Нахождение подобного рода ошибок возможно только с помощью так называемых *проверяющих достоверность* анализаторов. Документ считается *достоверным,* если он прошел проверку, определенную в схеме объявления DTD. Языки и приложения, основанные на XML, часто используют информацию из DTD для установки какого-либо минимального стандарта при распределении элементов и данных. Например, консорциум W3C зарегистрировал по крайней мере одно объявление DTD, в котором описывается XHTML (XML-совместимый тип HTML). В этом документе перечисляются все элементы, их внешний вид, целевое местоположение, а также их содержимое. Грамматически корректно можно было бы включить элемент <p> внутри тела документа (<body >), а вот, например, помещение элемента <p> внутри заголовка (<head>) будет уже некорректным. Тем более не следует включать элемент <blooby> где-нибудь в документе, поскольку он не описан в объявлении DTD[2]. Если в документе появится хотя бы одна ошибка такого типа, весь документ будет рассматриваться как *некорректный.* Он может подчиняться правилам формальной корректности, но не действительным но причине частичного противоречия с правилами, определенными в объявлении DTD. Часто такой уровень проверки больше мешает, чем помогает, но его наличие «греет душу».

В конце нашего списка находится требование, говорящее о необходимости отсылки анализатором переработанных данных программе или конечному пользова-

---

[1] Большинство HTML-браузеров пытаются игнорировать ошибки, связанные с нарушением формальной корректности документа. Подобный факт привычен для пользователя, наблюдающего перманентное снижение качества web-страниц, представленных и Сети. На самом деле лучше не допускать ошибки синтаксического разбора, чем пытаться их исправить в дальнейшем.

[2] Если вы захотите авторизовать и XML-коде web-страницы, использующие теги <blooby>, вы можете спроектировать свое собственное расширение путем разработки определения DTD. В этом документе определяется использование объектных ссылок для выборки информации в XHTML DTD. Затем в заголовке этого документа определяются пользовательские специальные элементы. В данном случае мы имеем дело с подклассом XHTML.

телю. В этом случае доступно несколько возможностей, анализу которых посвящены все последующие главы. Они образуют следующие категории:

- поток событий — анализатор может генерировать поток событий: поток символов разметки преобразуется в новый и более абстрактный тип потока, причем данные могут быть частично обработаны, а этот процесс упрощается вашей программой;
- представление объекта — также анализатор может конструировать структуру данных, которая отражает информацию в разметке XML. Подобная конструкция требует больше системных ресурсов, но может быть удобнее, поскольку в этом случае создается постоянный объект, который будет сохраняться на протяжении всего времени обработки;
- гибридная форма — мы могли бы назвать третью категорию «гибридным» выводом. Она включает анализаторы, которые придают процессу обработки более «интеллектуальный» характер, используя некоторые дополнительные знания о документе с целью конструирования объекта, представляющего только часть вашего документа.

Пример (которому не стоит следовать): проверка формальной корректности

До сих пор XML-анализаторы описывались несколько поверхностно, а сейчас пришло время расширить и углубить эту тему. Теперь мы приступим к написанию собственного анализатора, предназначенного для проверки формальной корректности документа. Этот анализатор является одним из простейших, которые только существуют в природе. С его помощью невозможно выполнять какую-либо дальнейшую обработку, а можно лишь давать однозначные ответы в стиле «да» или «нет».

В данном случае мы преследуем двоякую цель. Во-первых, мы надеемся немного приоткрыть «тайный занавес» над процессом XML-обработки — на самом деле речь идет еще об одном виде обработки текста. Однако мы также хотим подчеркнуть, что написание подходящего Perl-анализатора (как и анализатора для любого другого языка) требует немалых усилий, которые могут быть потрачены на создание более интересного кода, использующего один из многих доступных Perl-модулей, выполняющих синтаксический разбор. Для этого будет разработан фрагмент чистого XML-анализатора, предназначенного для реализации весьма специфической задачи.

ПРИМЕЧАНИЕ

Наслаждайтесь работой с этой программой, но, пожалуйста, не пытайтесь использовать этот код в профессиональной практике! В данном случае не идет речь о реальном интеграционном решении Perl и XML, а приводится лишь иллюстрация того, что может сделать подобный анализатор. Также, это решение неполноценно и не всегда дает адекватный результат (как будет показано в дальнейшем). Пусть вас не смущает это обстоятельство, поскольку в последующих главах описываются реальные XML-анализаторы и Perl-инструментарий.

Наша программа представляет собой цикл, в процессе выполнения которого регулярные выражения сопоставляются с объектами XML-разметки, а также отделяют их от текста. Этот цикл выполняется до тех пор, пока не завершатся действия по перемещению, т.е. пока документ не будет отвечать правилам формальной корректности либо пока регулярные выражения не будут соответствовать оставшимся в тек-

стс объектам (правила формальной корректности не соблюдаются). Процесс синтаксического разбора может быть остановлен в результате выполнения некоторых других проверок (например, если будет обнаружено, что имя завершающего тега не совпадает с именем открытого начального тега). Все это далеко от совершенства, по в итоге вы получите представление о том, как работает анализатор, проверяющий текст на соответствие правилам формальной корректности.

Листинг 3.1 — это процедура, которая анализирует строку XML-текста, то есть проверяет, проверяет соответствие текста правилам формальной корректности, а затем возвращает булево значение. С целью лучшего понимания сути регулярных выражений были добавлены некоторые переменные. Например, строка $ident содержит код регулярного выражения, позволяющего установить соответствие с XML-идентификатором, применяемым для элементов, атрибутов и инструкций по обработке.

Листинг 3.1. Элементарный XML-анализатор

```
sub is_well_formed {
    my $text = shift; # Проверяемый XML-текст.

    # Шаблоны для сравнения.
    my $ident = "[:_A-Za-z][:A-Za-z0-9\-\._]*";# Идентификатор.
    my Soptsp = "\s*"; # дополнительное пространство
    my $attl = «$ident$optsp=$optsp\»[^\»]*\»»; # Атрибут.
    my $att2 = «$ident$optsp=$optsp'[^']*'»;# Вариант атрибута.
    my $att = «($att1|$att2)»: # Любой атрибут.

    my @elements = ( ); # Стек открытых элементов.

    # Цикл по строке с целью выборки XML-разметки.
    while( length($text) ) {

    # Сравнение с пустым элементом.
    if( $text =~ /^&($ident)(\s+$att)*\s*\/>/ ) {
        $text = $';

    # Сравнение с начальным тегом элемента.
    } elsif( $text =~ /^&($ident)(\s+$att)*\s*>/ ) {
        push( @elements, $1 );
        $text = $';

    # Сравнение с завершающим тегом элемента.
    } elsif ( $text =~ /^&\/($ident)\s*>/ ) {
        return unless( $1 eq pop( @elements ));
        $text = $';

    # Сравнение с комментарием.
    } elsif( $text =~ /^&!—/ ) {
        $text = $ ' ;
        # Изъятие оставшейся части комментария.
        if( $text =~ /->/ ) {
            $text = $' ;
            return if( $" =~ /-/ ); # Комментарий не может содержать "-".
        } else {
            return;
        }
    # Сравнение с секцией CDATA.
    } elsif( $text =- /^&!\[CDATA\[/ ) {
        $text = $' ;
        # Исключение оставшейся части комментария.
```

```
        if( Stext =~ /\]\]>/ ) {
            $text = $';
        } else {
            return;
        }
    # Сравнение с инструкцией по обработке.
    } elsif( $text =~ m|^&\?$ident\s*[^\?]+\?> | ) {
        Stext = $' ;
    # Сравнение с дополнительными пробелами
    # (в случае, если пробел не включен в корневой элемент).
    } elsif( $text =~ m|^\s+| ) {
        Stext = $' ;
    # Сравнение с символьными данными.
    } elsif( $text =~ /(^[^&&>]+)/ ) {
        my $data = $1;
        # Проверка на нахождение данных внутри элемента.
        return if( $data =~ /\S/ and not( @elements ));
        $text = $';
    # Сравнение с объектной ссылкой.
    } elsif( Stext =~ /^&$ident;+/ ) {
        Stext = $' ;
    # Нечто непредвиденное.
    } else {
        return;
    }
    }
    return if( @elements ); # стек будет пустым
    return 1;
}
```

Полезность Perl-массивов частично объясняется их способностью к маскировке (как и в случае со многими другими абстрактными конструкциями данных)[1]. Здесь используется структура данных, называемая *стеком.* Эта структура на самом деле является просто массивом, доступ к которому определяется с помощью методов push () и POP (). Элементы в стеке расположены по принципу «последним пришел, первым ушел» (last-in, first-out, LIFO). Это означает, что когда последний элемент помещается в стек, то первый элемент оттуда удаляется. Такое устройство удобно для запоминания имен текущих открытых элементов, поскольку в любое время любой новый элемент закрывает предыдущий элемент, помещенный в стек. Всякий раз, когда мы сталкиваемся с начальным тегом, он будет помещен в стек, и он же удаляется из стека по достижении завершающего тега. Для того чтобы не нарушать правила формальной корректности, каждый завершающий тег должен соответствовать предыдущему начальному тегу (в этом случае стек оказывается незаменимым).

Стек представляет все элементы, расположенные вдоль ветви XML-дерева, от корня до текущего обрабатываемого элемента. Элементы обрабатываются в том порядке, в котором они находятся в документе. В случае представления документа в виде дерева, это будет выглядеть, будто вы идете от корня до конца одной ветви,

[1] Для получения дополнительных сведений по этой теме обратитесь к книге В. Водолазкого «Энциклопедия Perl» (издательство «Питер»).

затем назад к другой ветви и т.д.. Подобный способ называется *методом поиска в глубину.* Он является каноническим и применяется для обработки всех XML-документов.

Существует несколько моментов, проявляющихся в случае отступления от схемы простого цикла с целью выполнения нескольких дополнительных проверок. Код, проверяющий соответствие с комментарием, включает несколько шагов и завершается трехсимвольныш разделителем. Также необходима проверка наличия недопустимых пунктирных строк «—» внутри комментария. Программа проверки символьных данных, контролирующая степень заполненности стека, также требует внимания; если стек пуст, появляется сообщение об ошибке, поскольку вне корневого элемента не разрешается наличие текста, отличного от служебных символов. Ниже представлен короткий список ошибок, связанных с нарушением правил формальной корректности, распознаваемых анализатором:

- идентификатор элемента или атрибута некорректно сформирован (примеры: "12f roo," " -Ыa" и "..");
- символ, не являющийся пробелом, найден вне корневого элемента;
- завершающий тег элемента не соответствует обнаруженному начальному тегу;
- атрибут не упомянут или использует недопустимую комбинацию символов кавычек;
- пустой элемент пропускает символ слэша (/) в конце соответствующего ему тега;
- недопустимый символ типа одиночного амперсанда (&) пли угловой скобки (<) найден в символьных данных;
- найден некорректный тег разметки (примеры: "<fooby<" и "<? Bubba?>").

Испытаем наш анализатор в действии на нескольких тестовых примерах. Возможно, наипростейший завершенный XML-документ, отвечающий правилам формальной корректности, имеет такой вид:

Следующий документ приводит к прекращению действия анализатора при наличии ошибки. (Подсказка: обратите внимание на завершающий тег <message>.)

```
<memo>
     <to>self</to>
     <message>Don' t -forget to mow the car and wash the
     lawn.<message>
</memo>
```

В документе могут встречаться и другие виды синтаксических ошибок, и наша программа найдет большинство из них. Однако некоторые ошибки остаются необнаруженными. Например, в приведенном фрагменте кода должен указываться лишь один корневой элемент, но наша программа допускает наличие более чем одного элемента:

```
<root>I am the one, true root!</root>
<root>No, I am!</root>
<root>Uh oh...</root>
```

Итак, какие еще проблемы могут возникать в этом случае? Синтаксический анализатор не может обрабатывать объявление типа документа. Эта структура

иногда отображается в заголовке документа, который и определяет объявления DTD, используемые проверяющими достоверность анализаторами, а также может объявлять некоторые сущности. Используя присущий анализатору специализированный синтаксис, сконструируем еще один цикл, предназначенный для объявления типа документа.

Наиболее значительный недостаток нашего синтаксического анализатора коренится в методе, применяемом для разрешения объектных ссылок. Он может проверять базовый синтаксис объектных ссылок, но не способен расширять сущности и включать их в текст. Почему же так происходит? Считается, что сущность может содержать нечто большее, чем обычные символьные данные. Она может включать код разметки, который варьируется от отдельного элемента до большого внешнего файла. Сущности также содержат ссылки на другие объекты, так что может потребоваться несколько проходов для полного разрешения одной объектной ссылки. Анализатор даже не проверяет, объявлены ли сущности (эта операция невозможна в любом случае, поскольку недоступен метод, применяемый при чтении синтаксиса типов документов). При этом высока вероятность появления в документе ошибок, не обнаруживаемых анализатором. Для того чтобы избежать упомянутых проблем, следуйте указанным ниже правилам:

- добавьте цикл разбора с целью считывания объявления типа документа перед выполнением любого типа синтаксического разбора. Будут проанализированы и сохранены все объявления сущности документов, поэтому позднее можно разрешать объектные ссылки;
- в случае упоминания объявления типа документа выполните разбор DTD, в результате чего обеспечивается считывание любых объявлений сущности;
- в основном цикле разрешите все объектные ссылки по мере их появления. Эти объекты должны быть проанализированы наравне с присущими им объектными ссылками. Обработка должна быть циклической и включать несколько вложенных циклов, рекурсию и другие сложные программные конструкции.

И вот, простой анализатор превратился в сложного монстра! Отсюда можно сделать вывод, что теория разбора XML-кода проста только на первый взгляд, на практике же она усложняется очень быстро. Это упражнение было довольно полезным, поскольку оно явно продемонстрировало, что вовлечено в процесс синтаксического разбора, однако мы не одобряем написание подобных программ. Напротив, мы ожидаем, что вы воспользуетесь результатами исчерпывающей работы, уже проделанной при разработке синтаксических анализаторов.

Анализатор XML::Parser

Создание анализатора требует выполнения большого объема работы. Невозможно быть уверенным в том, что при написании программы учтены все моменты, пока не будет выполнено всестороннее тестирование кода. До тех пор пока вы не приобретете достаточный уровень квалификации, ваши программы будут медленными и потребляющими большой объем вычислительных ресурсов. Однако в настоящее время существует множество высококачественных, бесплатно распространяемых и простых в применении пакетов XML-анализаторов. Программисты, которые на протяжении многих лет занимались вопросами сопряжения Perl и XML,

разработали целый ряд технических приемок и «изюминок», значительно облегчающих вашу жизнь.

Каким же образом Perl-программисты могут получить доступ к готовым модулям с целью их применения в своих собственных программах? Проще всего обратиться ко Всеобъемлющей архивной сети Perl (Comprehensive Perl Archive Network, CPAN). Здесь можно получить свободно распространяемые Perl-модули, исходный код которых является открытым. Если вы до сих пор не пользовались сетью CPAN, настала пора изменить свое отношение к этому вопросу и интегрироваться в мировое сообщество программистов. В этой сети можно найти множество модулей, созданных опытными программистами на Perl и XML, решившими поделиться своими наработками с окружающим миром.

ПРИМЕЧАНИЕ

Не следует представлять сеть CPAN в виде каталога готовых программ, предназначенных для решения всех задач, связанных с XML-кодированием. Скорее это набор инструментов, своего рода фундамент, на основе которого можно создавать свои собственные программы. Некоторые модули носят специальный характер и ориентированы на работу в составе таких популярных XML-приложений, как RSS и SOAP. Большинство же модулей носят общий характер и могут применяться для решения других задач. Конечно, не исключена вероятность, что вы не найдете нужный модуль. В этом случае можно воспользоваться XML-модулями общего назначения, адаптировав их с целью решения конкретных задач. Процесс адаптации будет рассмотрен в дальнейшем.

Существует две основные категории различий между XML-анализаторами. Так, различается применяемый *стиль синтаксического разбора*, посредством которого выполняется обработка XML-кода. В частности, применяется несколько различных стратегий, таких как построение структуры данных или создание потока событий. Другой атрибут, присущий анализаторам, называется *степенью завершенности стандартов*. Спектр значений этого атрибута варьируется от узкоспециальных до исчерпывающих решений, основанных на различных стандартах. Причем баланс плавно перемещается в направлении от нетрадиционных решений к стандартным наработкам, по мере того как сообщество Perl-разработчиков реализует основные стандарты, такие как SAX и DOM.

Модуль `XML::Parser` является предшественником всех XML-процессоров, основанных на Perl. Этому анализатору присуще множество свойств, включая наличие различных стилей синтаксического разбора. Конечно, этот анализатор не совместим со многими стандартами, но до сих пор, в силу ряда причин, является широко распространенным. И этой популярности совершенно не мешает то, что `XML::Parser` использует нестандартные API и требует осторожности при осуществлении некоторых операций. Действительно этот анализатор обеспечивает приемлемое быстродействие и гибкость при разборе документов, но этим его преимущества и ограничиваются. Возможно, популярность этой программы связана с чувством первой любви, испытываемой многими людьми по отношению к событиям и вещам их юности. В любом случае, практически все модули и программы, имеющие отношение к миру Perl и XML, основаны на применении модуля `XML::Parser`.

Начиная с 2001 года стали появляться новые низкоуровневые модули-анализаторы, основанные на быстродействующих и поддерживающих многие стандарты библиотеках. В скором времени мы вкратце остановимся на этих модулях. Но начнем с рассмотрения модуля `XML::Parser`, отдав должное его заслугам и многочисленным функциональным свойствам.

Еще на заре возникновения XML программист по имени Джеймс Кларк (James Clark), воспользовавшись возможностями языка С, разработал библиотеку XML-анализатора под названием Expat[1]. Благодаря быстродействию, эффективности и стабильности этот анализатор вскоре стал стандартом «де-факто» для многих программистов, занятых проблемами интеграции XML и Perl. С целью облегчения задачи интеграции Ларри Уолл (Larry Wall) разработал низкоуровневую API-библиотеку, включив ее в состав модуля XML : : Parser : : Expat. Затем был надстроен еще один верхний уровень, получивший название XML : : Parser. Этот анализатор общего назначения мог применяться при решении различных задач. Благодаря поддержке Кларка Купера (Clark Cooper) анализатор XML : : Parser стал фундаментом многих разработанных XML-модулей.

Секрет успеха анализатора XML : : Parser заключается в применении возможностей языка С. В предшествующих параграфах уже рассматривалось создание простого анализатора, основанного на Perl. Если воспользоваться нашим примером с целью обработки большого XML-документа, это может привести к большим затратам времени. Конечно, полнофункциональные XML-анализаторы, созданные на основе Perl, могут переноситься на любую систему, но их быстродействие невелико. Более высокая производительность наблюдаются в случае скомпилированных на базе С анализаторов, таких как Expat. Причем, как и в случае с любым Perl-модулем, основанным на С-коде (благодаря наличию стандартной библиотеки Perl, XS[2], ничто не мешает широкому распространению подобных модулей), при использовании анализатора XML: : Parser не ощущается наличие Expat.

Пример: и снова программа проверки формальной корректности

С целью иллюстрации возможностей модуля XML: : Parser снова вернемся к проблеме проверки формальной корректности. Инструмент, пригодный для выполнения этой задачи, может быть легко создан с помощью анализатора XML: :Parser, как показано в листинге 3.2.

Листинг 3.2. Программа проверки формальной корректности, созданная на базе модуля XML::Parser

```
use XML::Parser;
my $xmlfile = shift @ARGV; # файл для синтаксического разбора
# Инициализация объекта анализатора и разбор строки.
my Sparser = XML::Parser->new( ErrorContext => 2 );
eval { $parser->parsefile( $xmlfile ); };
# Информация об ошибках, приведших к останову процесса
# синтаксического разбора, либо извещение об успехе.
if( $@ ) {
    $@ =~ s/at \/.*?$//s; # удаление номера строки модуля.
    print STDERR «\nERR0R in "$file":\n$@\n»;
```

[1] Имя Джеймса Кларка широко известно в среде разработчиков па языке XML. Он неутомимо утверждает свои стандарты в составе консорциума W3C, разрабатывая свободно распространяемые инструменты. Результаты его работ размещены па web-узле http://www.jclark.com/. Кларк также является автором рекомендационных документов но XSLT и XPath. Эти документы можно найти на web-узле http://www.w3.org/.

[2] Дополнительные сведения можно найти па man-странице команды perlxs.

```
} else {
    print STDERR «"$file" is well-formed\n»;
>
```

Приступим к рассмотрению структуры программы. Сначала создается новый объект XML: : Parser, предназначенный для выполнения синтаксического разбора. Благодаря тому что вместо вызова статической функции используется объект, можно сконфигурировать анализатор один раз, а затем обрабатывать различные файлы. При этом не требуется повторное создание этой программы. Во время выполнения синтаксического разбора файлов остаются неизменными все пользовательские настройки, а также не затрагивается анализатор Expat. После завершения работы производится очистка.

Затем в составе блока eval вызывается метод parsefile().Это связано с тем, что анализатор XML: : Parser проявляет излишнюю «ретивость» при обработке ошибок, выявленных на этапе синтаксического разбора. Если же блок eval не используется, выполнение программы может завершиться досрочно. И причиной этого может стать выполнение очистки. В случае возникновения какой-либо ошибки проверяется содержимое переменной $@. Если имеет место ошибка, производится удаление соответствующей строки модуля, а затем выводится на печать соответствующее сообщение. При инициализации объекта анализатора устанавливается опция ErrorContext => 2. Анализатор XML: : Parser включает несколько опций, позволяющих установить контроль процесса синтаксического разбора. В качестве одной из таких опций можно рассматривать директиву, отсылаемую непосредственно анализатору Expat. Эта директива используется с целью запоминания содержимого, в котором произошла ошибка, а также для сохранения двух строк, предшествующих ошибке. Затем выводится на печать сообщение об ошибке, в котором указывается номер строки, содержащей ошибку, а также отображается область текста со стрелкой, указывающей па саму ошибку.

Ниже приводится пример сообщения об ошибке, отображаемого программой проверки синтаксиса (наша программа называется xwf, аббревиатура, образованная от слов «XML well-formedness» (формальная корректность кода XML)):

```
$ xwf ch01.xml
ERROR in 'ch01.xml':
not well-formed (invalid token) at line 66, column 22, byte 2354:
<chapter id="dorothy-in-oz">
<title>Lions, Tigers & Bears</title>
=====================^
```

Обратите внимание, насколько просто было установить анализатор и получить нетривиальные результаты. Для оценки быстродействия программы необходимо запустить ее на выполнение. В частности, при вводе какой-либо команды результат отображается за долю секунды.

Рабочее конфигурирование анализатора может выполняться различными путями. Например, может потребоваться запретить выполнение синтаксического разбора файлов. Если необходимо произвести синтаксический разбор текстовых строк, воспользуйтесь методом parse(). Также может потребоваться опция NoExpand => 1, заменяющая статгдартное расширение сущности процедурой вашего собственного определителя сущностей. Эта опция может использоваться для предотвращения открытия внешних сущностей, ограничивая тем самым область проверки синтаксиса, осуществляемой анализатором.

Несмотря на то, что программа проверки формальной корректности является весьма полезным инструментом, при частой работе с XML-файлами его возможностей будет явно недостаточно. В следующем разделе рассматривается, какую важную роль играет анализатор при передаче данных в программу. При этом подчеркивается важность правильного выбора стиля синтаксического разбора.

Стили синтаксического разбора

Анализатор XML: : Parser поддерживает несколько различных стилей синтаксического разбора, применимых для реализации различных стратегий разработки. Используемый стиль не изменяет порядок считывания анализатором XML-кода. Скорее он переопределяет формат представления результатов синтаксического разбора. И если вы нуждаетесь в непротиворечивой структуре, предназначенной для хранения документа, вы ее получите. А если вы предпочитаете иметь анализатор, способный вызывать набор пользовательских процедур, то и в этом случае особых проблем не возникнет. Настройка стиля достигается путем инициализации объекта, осуществляемой присваиванием значения атрибуту style. Далее производится краткий обзор доступных стилей:

- *отладка (debug)* — при использовании этого стиля производится вывод документа в поток STDOUT, причем документ отформатирован в виде структуры (вложенные элементы располагаются с отступом). Метод parse () не возвращает программе какие-либо полезные данные;
- *дерево (tree)* — этот стиль создает иерархическую структуру данных в форме дерева, которое может обрабатываться программой. Эта форма содержит все элементы и их данные, причем ее структура включает вложенные хэш-объекты и массивы;
- *объект (object)* — как и в случае с деревом, возвращается ссылка на иерархическую структуру данных, которая представляет документ. Однако в этом случае вместо использования простых агрегатов данных, подобных хэш-объектам и спискам, применяются объекты, включающие XML-разметку;
- *процедуры (subs)* — этот стиль позволяет устанавливать *функции обратного вызова,* предназначенные для обработки отдельных элементов. В этом случае создается пакет процедур, вызываемых после обрабатываемых ими элементов. Затем анализатору сообщается о наличии подобного пакета с помощью опции pkg. Всякий раз, когда анализатор встречает начальный тег элемента <fooby>, начинается поиск метода fooby() внутри пакета. После того как анализатор встречает завершающий тег элемента, предпринимается попытка вызова функции _fooby(), входящей в состав пакета. При этом анализатор передает критически важную информацию, например, ссылки па содержимое и атрибуты функции. Благодаря этому становится возможным выполнение обработки со стороны пользователя;
- *поток (stream)* — Как и в случае со стилем Subs, можно определять обратные вызовы, реализующие обработку конкретных XML-компонентов, причем обратные вызовы носят более общий характер, чем имена элементов. Можно создавать функции, именуемые *обработчиками,* которые вызываются так называемыми «событиями». В роли события может выступать начало элемента (любой элемент, а не просто определенный сорт таких элементов),

набор символьных данных либо инструкция по обработке. Регистрация пакета обработчика производится с помощью опции Handlers либо метода setHandlers();

- *Настраиваемый (Custom)* — можно образовать подкласс класса XML: :Parser на основе пользовательского объекта. Подобная операция довольно часто применяется при создании API в стиле анализаторов, предназначенных для более специфичных приложений. Например, подобная стратегия может применяться модулем XML: : Parser: :PerlSAX с целью реализации стандарта обработки событий SAX.

Листинг 3.3 представляет собой образец программы, использующей анализатор XML: : Parser. Здесь атрибуту Style присвоено значение Tree. Находясь в этом режиме, анализатор считывает весь XML-документ в процессе построения структуры данных. После завершения этой работы устанавливается связь между программой и соответствующей ей структурой.

Листинг 3.3. Генератор XML-дерева

```
use XML::Parser;

# Инициализация анализатора и считывание файла.
Sparser = new XML::Parser( Style => "Tree" );
my $tree = $parser->parsefile( shift @ARGV );

# Преобразование структуры в последовательную форму.
use Data::Dumper;
print Dumper( $tree );
```

При переключении в режим дерева метод parsefileO возвращает ссылку на структуру данных, содержащую документ, который кодируется с применением хэш-объектов и списков. Модуль Data: : Dumper применяется для преобразования в последовательную форму структур данных, а также для просмотра результатов работы. Листинг 3.4 описывает применяемый файл данных.

Листинг 3.4. Файл данных XML

```
<preferences>
     <font role="console">
          <fname>Courier</name>
          <size>9</size>
     </font>
     <font role="default">
          <fname>Times New Roman</name>
          <size>14</si ze>
     </font>
     <font role="titles">
          <fname>Helvetica</name>
          <size>10</size>
     </font>
</preferences>
```

На основе подобного файла данных программа генерирует следующий вывод (соответствующим образом отформатированный с целью улучшения восприятия):

```
$tree = [
            'preferences' , [
              {}, 0, '\n' ,
              'font', [
                { 'role' => 'console' }, 0, '\п',
```

```
      'size' , [ {}, 0, '9' ], 0, '\n',
      'fname', [ {>, 0, 'Courier' ], 0, '\n'
    ], 0, '\n',
    'font', [
      { 'role' => 'default' }, 0, '\n',
      'fname', [ {}, 0, 'Times New Roman' ], 0, '\n',
      'size', [ {} , 0, '14' ] , 0, '\n'
    ], 0, '\n',
    'font', [
      { 'role' => 'titles' } , 0, '\n' ,
      'size', [ {}, 0, '10' ] , 0, '\n',
      'fname', [ {}, 0, 'Helvetica' ], 0, '\n',
    ] , 0, '\n',
  ]
];
```

Значительно проще создать код, «анатомирующий» указанную выше структуру, чем написать собственный анализатор. С другой стороны, анализатор возвращает структуру данных вместо малопонятного промежуточного продукта синтаксического разбора, благодаря которой можно сделать вывод о стопроцентном соответствии документа правил формальной корректности XML. В главе 4 подробнее рассматривается потоковый режим XML::Parser, а в главе 6 мы углубимся в «дебри» деревьев и объектов.

Два различных подхода к обработке данных: деревья и потоки

Помните основное правило, справедливое для Perl: «Существует более одного способа для выполнения одного и того же действия»? Это правило также остается справедливым в мире XML. Пользователю доступны различные возможности, в зависимости от поставленной цели и ресурсов, которыми он располагает. Одни разработчики предпочитают выполнение синтаксического разбора, поскольку эта работа является относительно простой. При этом они готовы идти па издержки, связанные с избыточным потреблением памяти и других системных ресурсов. Другим требуется улучшенная производительность и экономия системных ресурсов за счет создания более сложного кода.

Обработка XML-кода может осуществляться с применением различных стратегий. Большинство из них относятся к следующим двум категориям: основанные на потоках и основанные на деревьях. Использование *стратегии, основанной на потоках,* подразумевает непрерывную передачу информации программе анализатором в соответствии с XML-шаблонами. Анализатор функционирует подобно каналу, в качестве исходных данных для которого выступает XML-разметка. Затем обработанные фрагменты данных передаются программе. Этот капал называется *потоком событий.* Происхождение подобного названия связано с тем, что каждый фрагмент данных, отсылаемый программе, сигнализирует о чем-то новом и интересном, происшедшем в XML-потоке. Например, к разряду важных событий можно отнести начало нового элемента. В этом случае происходит поиск инструкций по обработке, находящихся в тексте разметки. В результате выполнения каждого нового обновления программа осуществляет новую трансляцию данных, производя их передачу в новое местоположение. Затем выполняется тестирование на

предмет наличия специфического содержимого, которое может помещаться в постоянно растущую «кучу» данных.

В рамках *стратегии, основанной на деревьях,* анализатор сохраняет данные до тех пор, пока не будет создана завершенная модель документа. В данном случае идет речь не о канале, а о камере, с помощью которой делаются снимки, передаваемые пользователю. Эта модель более удобна в применении, чем исходный XML-код. Например, вложенные элементы могут представляться к виде собственных Perl-структур, подобных спискам либо хэш-объектам, как упоминалось в предыдущем примере. Еще более уместным выглядит применение деревьев при работе с объектами, которым присущи методы, облегчающие перемещение в пределах структуры. Ключевое преимущество этой стратегии проявляется в том, что программа может производить быструю выборку данных в любом порядке.

Итак, какую из описанных стратегий следует предпочесть? Каждая из них обладает определенными преимуществами и недостатками. Потокам событий присуще быстродействие и малое потребление ресурсов памяти. За это приходится расплачиваться более сложным кодом и непостоянством данных. С другой стороны, при построении деревьев становится возможным фиксация данных до тех пор, пока они будут нужны. Код, применяемый в этом случае, обычно проще. Это связано с отсутствием специальных приемов, предназначенных для реализации таких методик, как, например, обратный поиск. Деревья применять не следует в случае ограниченности ресурсов машинного времени и памяти.

Конечно, все эти соображения носят относительный характер. Обработка небольших документов не представляет особых затруднений для типичного компьютера, особенно если учесть тенденцию постоянного удешевления аппаратных компонентов. Вполне возможно, что удобство совместимой структуры данных перевешивает другие недостатки. С другой стороны, при работе с огромными документами, либо при одновременной обработке большого числа документов, могут возникать определенные проблемы. В этом случае большую ценность приобретает быстродействие обработчиков потока событий. Невозможно выработать рецепты, пригодные на все случаи жизни, поэтому окончательное решение остается за вами.

Интересно отметить, что на базе стратегии, основанной на деревьях, может разрабатываться стратегия, основанная на потоках. Также справедливо обратное утверждение. Так, например, потоки событий управляют процессом построения структур данных, организованных в виде деревьев. Таким образом, в качестве анализаторов самого нижнего уровня могут применяться потоки событий, поскольку над ними можно всегда надстроить уровень, реализующий создание деревьев. Именно на этом основана работа XML :: `Parser` и большинства других анализаторов.

В процессе разработки современных программ потоки событий XML могут включать любой тип документа в состав некоторых форм XML-кода. На практике это достигается путем создания анализаторов, основанных на потоках. Эти анализаторы генерируют XML-события на основе произвольных структур данных, скрывающихся в данном типе документа.

Можно еще многое сказать о потоках событий и построителях деревьев, но не в этой главе. Фактически данная тема обсуждается в двух отдельных главах этой книги. В главе 4 приводится теория, лежащая в основе потоков событий, а также изложены многочисленные практические примеры. В главе 6 рассматриваются де-

ревья и соответствующие практические примеры. В главе 8 демонстрируются своего рода «гибриды», воплощающие лучшие качества описанных структур данных.

Практическое применение анализаторов

На этом завершим обсуждение подробностей внутреннего устройства анализаторов. Наступило время перейти к практическим аспектам их применения. До сих пор мы уже рассматривали пример завершенной структуры дерева, построенной с помощью анализатора (пример 3.3). А теперь приступим к рассмотрению других структур данных. Возьмем поток XML-события и задействуем в процессе обработки путем его включения в состав некоторого кода, предназначенного для выполнения обработки событий. Конечно, этот инструмент нельзя назвать наилучшим в мире, но его возможностей будет вполне достаточно, чтобы продемонстрировать пользователю методы разработки реальных программ обработки из мира XML.

В качестве точки входа программы будет применяться анализатор XML: : Parser (причем на нижнем уровне выполняется Expat). Анализатор Expat имеет отношение к упомянутой ранее методике синтаксического разбора, основанного па событиях. Вместо того чтобы загружать весь XML-документ в память с последующим его подробным обзором, в данном случае происходит останов всякий раз, когда встречается дискретный фрагмент данных либо разметки, например, тег, заключенный в угловые скобки, либо строка из литералов, находящаяся внутри элемента. Затем проверяется реакция программы на наличие подобных событий.

Вашей первой обязанностью является предоставление анализатору интерфейса, реализованного в виде уместных в данной ситуации фрагментов кода. Каждый тип события обрабатывается с помощью отдельной процедуры, также называемой обработчиком. Регистрация обработчиков вместе с анализатором осуществляется с помощью установки значения опции Handlers во время инициализации. Весь описанный процесс иллюстрируется листингом 3.5.

Листинг 3.5. XML-процессор, основанный на потоках

```
use XML::Parser;
# Инициализация анализатора.
my Sparser = XML::Parser->new( Handlers =>
                                         {
                                          Start=>\&handle_start,
                                          End=>\&handle_end,
                                         });
$parser->parsefile( shift @ARGV );
my @element_stack;                # Помните об открытых элементах.

# Обработка события начало-элемента: вывод на печать
# сообщения об элементе.
#
sub nandle_start {
    my( $expat, $element, %attrs ) = @_;

    # Запрос объекта expat о нашей позиции.
    my $line = $expat->current_line;
```

```
        print "I see an $element element starting on line $line!\n";
        # Запоминание элемента и его начальной позиции путем
        # помещения небольшого хэша в стек элементов.
        push( @element_stack, { element=>$element, line=>$line }):
        if( %attrs ) {
            print "It has these attributes:\n" ;
            while( my( $key, lvalue ) = each( %attrs )) {
                print "\t$key => $value\n";
            }
        }
    }

    # Обработка события конца элемента.
    #
    sub handle_end {
        my( $expat, $element ) = @_;
        # Мы просто выталкиваем элемент из стека,
        # встретив корректный завершающий элемент, в отличие от
        # ситуации с доморощенной формальной корректностью, когда
        # XML::Parser "громко возмущается" при наличии
        # ошибок, связанных с нарушением формальной корректности
        my $element_record = pop( @element_stack );
        print "I see that $element element that started on line ",
              $$element_record{ line }, " is closing now.\n";
    }
```

Разобраться в том, как выполняется этот процесс, не составляет никакого труда. Просто были разработаны две процедуры обработчика, handle_start() и handle_end(), причем каждая из них регистрируется вместе с конкретным событием при вызове метода new(). При использовании метода parse() анализатор «знает» о наличии обработчиков событий начала и конца элемента. Каждый раз, когда анализатор находит начальный тег элемента, вызывается первый обработчик, которому передаются имя и атрибуты элемента. Таким же образом, при обнаружении завершающего тега вызывается другой обработчик, которому также передастся специфичная для элемента информация.

Обратите внимание на то, что анализатор передает каждому обработчику ссылку под именем $expat. Эта ссылка устанавливается на объект XML::Parser::Expat, интерфейс нижнего уровня для Expat. Подобная ссылка обеспечивает доступ к информации, которая может оказаться полезной для программы (например, номера строк и глубина вложенности элемента). Вы также можете воспользоваться подобными сведениями с целью выполнения анализа документов.

Не хотите ли посмотреть на пример выполнения программы? Далее приводится результат обработки документа базы данных заказчиков, приведенной в листинге 1.1:

```
I see a spam-document element starting on line 1!
It has these attributes:
        version => 3.5
        timestamp => 2002-05-13 15:33:45
I see a customer element starting on line 3!
I see a first-name element starting on line 4!
I see that the first-name element that started online4 is closing now.
I see a surname element starting on line 5!
```

```
I see that the surname element that started on line5 is closing now.
1 see a address element starting on line 6!
I see a street element starting on line 7!
I see that the street element that started on line 7 is closing now.
I see a city element starting on line 8!
I see that the city element that started on line 8 is closing now.
I see a state element starting on line 9!
I see that the state element that started on line 9 is closing now.
I see a zip element starting on line 10!
I see that the zip element that started on line 10 is closing now.
1 see that the address element that started on line6 is closing now.
I see a email element starting on line 12!
I see that the email element that started on line 12 is closing now.
I see a age element starting on line 13!
I see that the age element that started on line 13 is closing now.
I see that the customer element that started on line3 is closing now.
  [... snipping other customers for brevity's sake ...]
I see that the spam-document element that started on line 1 i s closing now.
```

В этом примере снова использовался стек элементов. На самом деле вовсе не требуется хранение имен элементов самих по себе. Один из методов, применяемых пользователем, заключаемся в вызове объекта XML : : Parser : : Expat. В результате этого возвращается перечень текущего содержимого, упорядоченный в хронологическом порядке перечень всех элементов, проверенных пользовательским анализатором. В любом случае применение стека обеспечивает хранение дополнительной информации например номеров строк. Это означает, что события могут создавать структуры произвольного уровня сложности в «памяти» о прошлом документа.

Подлежат обработке и многие другие типы событий. Например, ничего нельзя сделать с символьными данными, комментариями и инструкциями по обработке. Однако в рамках данного примера нам вовсе не требуется рассмотрение подобного типа событий. Тем более что в следующей главе приводятся многочисленные примеры обработки событий.

Прежде чем завершить тему обработки событий, упомянем еще об одном объекте — библиотеках Simple API, применяемых для XML-обработки (SAX). Используемая в этом случае технология напоминает рассмотренную ранее модель обработки событий, за одним лишь исключением. В данном случае применяется стандарт, одобренный консорциумом W3C. Благодаря этому обстоятельству стандартизируется канонический набор событий. Также внедряется метод обработки этих событий. В результате образуется стандартный интерфейс, позволяющий объединять различные программные компоненты с целью их совместного применения. Если вам не нравится применяемый анализатор, просто воспользуйтесь другим анализатором (либо такими сложными инструментами, как семейство модулей XML : : SAX , с помощью которых можно выбирать анализатор на основе требуемых свойств). При этом не имеет значения источник XML-данных (база данных, файл либо перечень покупок' вашей матушки). Инструмент SAX является весьма привлекательным для сообщества пользователей Perl, поскольку всем уже порядком надоело отсутствие совместимости со стандартами и всеобщи!! «вандализм». Теперь же поводов для критики стало заметно меньше. Подробное рассмотрение возможностей SAX (изначальное название PerlSAX) производится в главе 5.

Анализатор XML::LibXML

Анализатор XML::LibXML, подобно XML::Parser, представляет собой интерфейс библиотеки языка С. Эта библиотека является частью проекта GNOME[1] и называется libxml2. В отличие от XML::Parser, новый анализатор поддерживает базовый стандарт, применяемый при обработке XML-деревьев. Этот стандарт называется объектной моделью документа (Document Object Model, DOM).

Стандарт DOM — еще один пример шумно разрекламированного XML-стандарта. Он применяется для обработки деревьев, в то время как SAX предназначается для обработки потоков событий. Если вы склоняетесь к применению деревьев в своих программах и если допускаете вероятность повторного применения различных источников данных, воспользуйтесь стандартным решением с возможностью его изменения в дальнейшем. Дополнительные сведения о стандарте DOM и связанных с ним вопросах можно получить в главе 7.

Ну а теперь пришло время продемонстрировать в действии другой анализатор. Не стоит «зацикливаться» на одном анализаторе, каким бы хорошим он ни был. И снова будет продемонстрирован простейший пример, не представляющий собой ничего особенного. Задача этого примера будет заключаться лишь в вызове анализатора и демонстрации его в действии. Давайте разработаем другой инструмент разбора документов, подобный тому, который разрабатывался в листинге 3.5. На этот раз в задачи инструмента входит отображение частоты распределения элементов в документе.

Код разработанной программы приводится в листинге 3.(3. Здесь приводится простой пример анализатора, для которого не устанавливались опции. Фактически анализатор выполняет синтаксический разбор дескриптора файла и возвращает DOM-объект, представляющий собой древовидную структуру хорошо спроектированных объектов. Программа ищет элемент документа, а затем обходит все дерево одного элемента в то время, когда обновляются счетчики частоты распределения хэш-данных.

Листинг 3.6. Программа подсчета частоты распределения

```
use XML::LibXML;
use IO::Handle;
# Инициализация анализатора.
my Sparser = new XML::LibXML;
# Открытие дескриптора файла и синтаксический разбор.
my $fh = new IO::Handle;
if( $fh->fdopen( fileno( STDIN ), "r" )) {
    my $doc = $parser->parse_fh( $fh );
    my %dist;
    &proc_node( $doc->getDocumentElement, \%dist );
    foreach my $item ( sort keys %dist ) {
       print "$item: ", $dist{ $item }, "\n";
    }
    $fh->close;
}
# Обработка узла XML-дерева: если идет речь об элементе,
```

[1] Для загрузки модуля анализатора и комплекта сопровождающей документации обратитесь к web-узлу http://www.libxml.org/.

```
# обновляется список распределения и обрабатываются
# все потомки.
#
sub proc_node {
    my( $node, $dist ) = @_;
    return unless( $node->nodeType eq &XML_ELEMENT_NODE );
    $dist->{ $node->nodeName } ++;
    foreach my $child ( $node->getChildnodes ) {
        &proc_node( $child, $dist );
    }
}
```

Обратите внимание, что вместо указания простого пути к файлу, используется объект дескриптора файла из класса IO::Handle. Возможно, вы уже знаете, что дескрипторы Perl-файлов являются «шустрыми и магическими бестиями», передающими программе символы из самых различных источников: файлов на диске, открытых сетевых сокетов, потоков данных, вводимых с клавиатуры, баз данных, а также всех прочих источников данных. Как только был определен источник дескриптора файлов, в распоряжении пользователя оказывается интерфейс, обеспечивающий выполнение аналогичных операций считывания для любого дескриптора файла. Это обстоятельство идеально соответствует используемой нами XML-идеологии, когда при разработке кода ставится задача достижения максимальной степени гибкости и способности к повторному применению. В результате XML-код не «заботится» о своем происхождении, а у программиста отсутствует возможность его привязки к какому-либо тину источника.

После завершения стадии синтаксического разбора анализатор возвращает объект документа. Этот объект включает метод, который возвращает ссылку на элемент документа. В данном случае элемент документа представляет собой корень всего дерева. Программисты используют эту ссылку в качестве исходной для рекурсивной процедуры, proc_node(). Эта процедура обрабатывает элементы, внося запись в хэш-переменную всякий раз, когда встречается какой-либо элемент. Полезность рекурсии при написании программ, обрабатывающих XML-код, мотивируется тем, что структура документов напоминает фрактал: одни и те же правила применяются на любом уровне либо позиции в документе, включая корневой элемент, представляющий весь документ. Обратите внимание на проверку «типа узла», которая выделяется среди элементов и других частей документа (например, части текста или инструкций по обработке).

При нахождении процедурой любого элемента производится вызов метода объекта, getChildnodes(), обрабатывающего потомки этого элемента. Именно в факте наличия подобного вызова коренится различие между методологиями, основанными на потоках и деревьях. Вместо того чтобы сделать поток событий «ведущим колесом» программы и передавать ему данные для обработки, осуществляется вызов процедур и блоков кода в некотором непредсказуемом порядке. Причем программа несет ответственность за выполнение навигации в пределах документа. Традиционно начало обработки определяется корневым элементом, затем обработка распространяется далее. Обработка потомков производится в направлении от первого к последнему элементу. И поскольку мы не находимся под контролем анализатора, можем просматривать документ любым способом. Возможен обратный проход, просмотр части документа, либо многочисленные обходы дерева. Ниже приводится результат обработки небольшой главы, находящейся в XML-документе DocBook:

```
$ xfreq < ch03.xml
chapter: 1
citetitle: 2
firstterm: 16
footnote: 6
foreignphrase: 2
function: 10
itemizedlist: 2
listitem: 21
literal: 29
note: 1
orderedlist: 1
para: 77
programlisting: 9
replaceable: 1
screen: 1
section: 6
sgmltag: 8
simplesect: 1
systemitem: 2
term: 6
title: 7
variablelist: 1
varlistentry: 6
xref: 2
```

В результате отображается всего лишь несколько строк кода, но за этим скрывается большой объем работы. Опять-таки, благодаря использованию базовой С-библпотеки код получается достаточно быстрым.

Анализатор XML::XPath

До сих пор рассматривались примеры анализаторов, предназначенных для обработки всего документа. Однако зачастую в этом нет никакой нужды. Например, при выполнении запроса к базе данных обычно идет поиск только одной записи. При «взломе» телефонной книги вряд ли вам потребуется вся хранящаяся в ней информация. Следовательно, требуется некий механизм, определяющий выборку специфичной информации из обрабатываемого документа. В этом случае идеальным будет инструмент XPath.

Выбор этого инструмента рекомендуется разработчиками языка XML[1]. Он является базовым при создании выражений, применяемых для указания специфичных частей документа. Можете себе представлять все это в виде некой схемы адресации. И хотя подробности и детали применения инструмента XPath будут приведены в главе 8, сейчас просто скажем о том, что все это работает как и при совместном использовании регулярных выражений и путей к файлам в стиле Unix. Неудивительно, что это привлекательное свойство желательно включат!) в состав анализаторов.

Модуль XML::XPath, разработанный Мэттом Сержантом (Matt Sergeant), представляет собой основательную реализацию, созданную на основе XML::Parser. Получая XPath-выражения, он возвращает перечень всех частей документа, сопоста-

[1] Обратитесь к соответствующей спецификации по адресу http://www.w3.org/TR/xpath/.

вимых с описанием. При этом обеспечивается весьма простой способ выполнения сложных операций поиска и выборки,

Например, пусть в вашем распоряжении имеется адресная книга, созданная с помощью базовой формы XML-кода:

```
<contacts>
    <entry>
        <name>Bob Snob</name>
        <street>123 Platypus Lane</street>
        <city>Burgopolis</city>
        <state>FL</state>
        <zip>12345</zip>
        </entry>
    <!—Здесь могут быть дополнительные записи->
</contacts>
```

Допустим, что требуется осуществить выборку из файла всех почтовых индексов и скомпилировать их в виде списка. В листинге 3.7 показано, каким образом все это делается с помощью анализатора XPath.

Листинг 3.7. Программа-экстрактор почтовых индексов

```
use XML::XPath;

my $file = 'customers.xml';
my $xp = XML::XPath->new(filename=>$file);

# Набор узлов XML::XPath представляет собой объект,
# содержащий результаты оценивания XML-документа с
# применением Xpath-выражений: мы просто собираемся
# сделать это, а затем запросить набор узлов
# для просмотра того, что мы можем получить.
my $nodeset = $xp->find('//zip');

my @zipcodes;              # Здесь помещены результаты
if (my @nodelist = $nodeset->get_nodelist) {
  # Мы нашли почтовые индексы! Каждый узел — объект
  # класса XML::XPath::Node::Element. поэтому мы
  # воспользуемся методом класса 'string_value'
  #  для выборки подходящего текста, а результаты
  # выборки для всех узлов помещаются в массив.
  @zipcodes = map($_->string_value , @nodelist);
  # Сортировка и подготовка для вывода
  @zipcodes = sort(@zipcodes);

  local $" = "\n";
  print "I found these zipcodes:\n@zipcodes\n" :
} else {
  print "The file $file didn't have any 'zip' elements in it!\n";
}
```

Запустите программу на выполнение, воспользовавшись документом, содержащим три записи, В результате будет получено следующее:

```
I found these zipcodes:
03642
12333
82649
```

Этот модуль также демонстрирует пример синтаксического разбора, основанного на деревьях. Причем в этом случае анализатор загружает весь документ в дерево объекта, имеющего специфическую конструкцию. Затем пользователю раз-

решается избирательно взаимодействовать с частями дерева с помощью XPath-выражений. Рассматриваемый пример является лишь образцом того, что можно получать с модулями расширенной обработки деревьев. Дополнительные сведения относительно этих модулей можно найти в главе 8.

Объекты элемента XML : : LibXML поддерживают метод findnodes(), функционирующий аналогично XML: :XPath. При этом в качестве текущего содержимого применяется вызов объекта Element и возвращается список объектов, соответствующих запросу. Эти функциональные свойства рассматриваются в главе 10.

Проверка достоверности документа

Соответствие требованиям формальной корректности является необходимым условием при обеспечении корректности XML-кода. При этом XML-процессоры многое просто не проверяют. Если попытаться создать документ, удовлетворяющий некоторым спецификациям, указанным для XML-приложений, могут возникнуть затруднения. Причем затруднения возникают в том случае, если генератор содержимого пропускает необычный элемент, который никогда не встречался ранее, а анализатор не обращает на это ни малейшего внимания. К счастью при проверке в подобном случае доступен высший уровень контроля — проверка достоверности документа.

Проверка достоверности представляет собой сложный процесс сравнения экземпляра документа с шаблоном либо спецификацией грамматики. При этом может ограничиваться количество и тип используемых элементов документа, а также контролируется их местоположение. Можно даже настраивать шаблоны символьных данных для любого элемента либо атрибута. Проверяющий анализатор сообщает о том, будет ли корректным данный документ в случае наличия схемы либо шаблона DTD, с которыми и производится сравнение.

Необходимо помнить о том, что не стоит осуществлять проверку достоверности любого XML-документа. Объявления DTD и схемы проверки достоверности хороши в случае работы со специфическими языками разметки, основанными на XML (XHTML при работе с web-страницами, MathML при работе с математическими уравнениями либо некоторыми другими языками). При этом применяются ограниченные правила по отношению к объектам и атрибутам.

Проверка достоверности документа не требуется в случае, если возможности Perl и XML используются для выполнения менее специфичных задач, таких как быстрое объединение и анализ существующих XML-документов, использование экзотических форматов данных.

Как правило, если вы ощущаете потребность в выполнении проверки документа, лучше прислушаться к своим ощущениям. Операция проверки документа является обязательным условием в случае учета некоторых особенностей XML, требуемых в целях достижения совместимости с определенным стандартом. Ваш набор инструментов обеспечивает множество способов для выполнения проверки документов.

Объявления DTD

Объявления типов документа (Document type description, DTD) — документы, написанные с применением специального языка разметки, определенного в XML-

спецификации. Тем не менее, эти объекты не относятся к XML. Все, что находится внутри этих документов, называется объявлениями. Все объявления начинаются разграничителем <! и включают четыре области: элементы, атрибуты, сущности и записи.

В листинге 3.8 содержится описание простого объявления DTD.

Листинг 3.8. Небольшое объявление DTD

```
<!ELEMENT memo (to, from, message)>
<!ATTLI5T memo priority (urgent|normal|info) 'normal'>
<!ENTITY % text-only "(#PCDATA)*">
<!ELEMENT to %text-only;>
<!ELEMENT from %text-only;>
<!ELEMENT message (#PCDATA | emphasis)*>
<!ELEMENT emphasis %text-only;>
<!ENTITY myname "Bartholomus Chiggin McNugget">
```

В приведенном примере DTD объявляются пять элементов, атрибут элемента <memo>, параметр сущности, упрощающий другие объявления, а также сущность, которая может использоваться внутри экземпляра документа. Проверяющий анализатор может одобрять или отклонять документ на основе этой информации. Рассмотрим следующий пример:

```
<!DOCTYPE memo SYSTEM "/dtdstuff/memo.dtd">
<memo priority="info">
     <to>Sara Bellum</to>
     <from>&myname;</from>
     <message>Stop reading memos and get back to work!</message>
</memo>
```

Если из документа удалить элемент <to>, это может привести к нарушению корректности документа. Программа проверки формальной корректности не способна обнаруживать отсутствие подобных элементов. Потому в этом случае понадобится выполнение проверки документа.

Известно, что синтаксический разбор объявлений DTD не составляет особого труда. Учитывая это обстоятельство, некоторые XML-процессоры общего назначения могут осуществлять проверку достоверности документов на основе синтаксического разбора соответствующих объявлений DTD. Анализатор XML: : LibXML относится к подобным программам. Код простого анализатора, выполняющего проверку достоверности, приводится в листинге 3.9.

Листинг 3.9. Анализатор, выполняющий проверку достоверности

```
use XML::LibXML;
use IO::Handle;

# Инициализация анализатора.
my Sparser = new XML::LibXML;

# Открытие файлового дескриптора и разбор.
my Sfh = new IO::Handle:
if( $fh->fdopen( fileno( STDIN ), "r-" )) {
    my $doc = $parser->parse_fh( $fh );
    if( $doc and $doc->is_valid ) {
        print "Yup, it's valid.\n";
    > else {
        print "Yikes! Validity error.\n";
    }
```

```
    $fh->close;
}
```

Описываемый анализатор может быть легко встроен в любую программу, требующую проверки достоверности входных документов. К сожалению, в этом случае не предоставляется какая-либо информация, конкретизирующая возникшую проблему (например, элемент, находящийся на неподходящем месте). По этой причине подобный анализатор не может использоваться в качестве инструмента проверки достоверности общего назначения[1]. Лучше воспользоваться анализатором XML::Checker, разработанным Т. Дж. Мэттером (T. J. Mather). В результате его применения генерируются отчеты о специфических ошибках, связанных с проверкой достоверности данных.

Схемы

Объявления DTD имеют свои ограничения, в силу которых невозможно проверить тип символьных данных, назначенных элементу, а также факт совпадения этого элемента с тем либо иным шаблоном. Что же делать, если анализатор должен сообщать о том, что элемент <date> поддерживает некорректный формат даты либо включает неправильный адрес? В этом случае придется прибегнуть к решению под названием XML Schema. Документ XML Schema можно охарактеризовать как объявление DTD второго поколения, обладающее дополнительными возможностями и гибкостью, реализуемыми в процессе проверки документов.

В главе 2 уже упоминалось о том, что XML Schema слывет одной из наиболее противоречивых схем (по крайней мере, среди хакеров) в семействе XML-программ, определенных в спецификации W3C. Большинство пользователе!! совсем не против концепции схем как таковых, но им не нравится реализация XML Schema в силу се громоздкости и невозможности эффективного применения.

В качестве альтернативы XML Schema можно рассматривать RelaxNG, разработанный OASIS-Open (http://www.oasis-open.org/committees/relaxng/), и Scheinatron, разработанный Риком Джелиффом (Rick Jelliffe) (http://www.ascc.net/xml/resource/schematron/schematron.html). Как и в случае с XML Schema, эти спецификации детализуют XML-языки, используемые для описания других XML-языков, При этом программа, взаимодействующая с той или иной схемой, может использовать ее для проверки достоверности других XML-документов. Программа Schematron особо интересна, поскольку в ее состав входит Perl-модуль, позволяющий использовать ее на временной основе (в форме семейства функций XML::Schematron, 'разработанного Кином Хэмптоном (Kip Hampton)).

Программа Scheinatron представляет особый интерес для хакеров, использующих возможности Perl и XML. Это связано с тем, что программа реализует надстройку существующих популярных XML-технологий, для которых уже имеются Perl-реализации. Она помогает при определении простейшего языка, с помощью

[1] Авторы книги предпочитают использовать инструмент, работающий в режиме командной строки, *nsgmls*, доступный на web-узле http://www.jdark.com. Проверка достоверности произвольных документов может также выполняться па общедоступных web-узлах, таких как http://www.stg.brown.edu/service/xmlnalid. Обратите внимание, что в этих случаях XML-документ должен включать объявление DOCTYPE. Причем системный идентификатор должен содержать разрешимую ссылку URL, отличную от пути, определенного в системе.

которого возможно перечисление и группирование конструкций, которые выглядят подобно XPath-выражспиям. Вместо опережающей грамматики, в которой перечисляется и определяется все, что может отображаться в документе, можно устанавливать достоверность лишь части структур документа. Можно проверять достоверность элементов и атрибутов на базе условий, охватывающих другие части документа (здесь идет речь о тех частях документа, которые «попадают в сферу влияния» XPath-выражепий). На практике документ, созданный с помощью Schematron, выглядит и ведет себя подобно таблице стилей XSLT. Благодаря этому обстоятельству документ может быть полностью реализован с помощью XSLT. Фактически функционирование двух Perl-модулей XML : : Schematron осуществляется путем первичного преобразования документа схемы, определенного пользователем, в XSLT-таблицу. Затем продукт преобразования просто передается XSLT-процессору.

В программе Schematron отсутствует типизация встроенных типов данных, поэтому, например, невозможно выполнить однословную проверку с целью обеспечения соответствия атрибута с форматом данных W3C. Однако Perl-программа может выполнить отдельный шаг, в ходе которого реализуется метод, определенный на ваше усмотрение (даже с помощью модуля XML : :XPath). Этот метод реализует просмотр атрибутов данных и выполнения регулярных Perl-выражений, в качестве аргументов которых выступают просматриваемые атрибуты. Обратите внимание на то, что языки схем не поддерживают методы запроса содержимого элемента в базе данных, а также не используются для выполнения любых действий, не предназначенных для обработки документа. Поэтому интеграция возможностей схем и Perl будет весьма полезной.

МодульXML::Writer

По сравнению с заданиями, которые выполнялись до сих пор в этой главе, написание XML-кода представляет собой простейшую задачу. Простота создания этого кода объясняется тем, что программа снабжена полностью контролируемой структурой данных. Поэтому не требуется предусматривать какие-либо случайности, происходящие при обработке входных данных.

Генерирование XML-кода не связано с какими-либо особыми проблемами. Пользователь располагает информацией об элементах, обладающих начальными и завершающими тегами, их атрибутами и прочими сведениями. Конечно, создание метода вывода XML-кода, учитывающего все подробности, представляется довольно скучным занятием, ведь потребуется ответить на целый ряд вопросов. Были ли вставлены пробелы между атрибутами? Были ли закрыты открытые элементы? Был ли включен слэш в конце пустых элементов? Как правило, подобными вопросами не задаются до тех пор, пока не создастся более или менее важный код. Еще один способ обработки подобных деталей заключается в создании специальных модулей.

Модуль XML : :Writer,созданный Дэвидом Мэггинсоном (David Megginson), является превосходным примером абстрактного интерфейса, обеспечивающего генерирование XML-кода. Этот интерфейс поставляется вместе с набором простых и полезных методов, обеспечивающих создание любого XML-докумепта. Просто создайте объект, предназначенный для написания кода, и вызовите его методы, обеспечивающие создание потока XML-кода. Некоторые из применяемых в данном случае методов перечислены в табл. 3.1.

Таблица 3.1 Методы модуля XML::Writer

Имя	Функция
end()	Закрывает документ и выполняет простейшую проверку формальной корректности (т.е. проверяется наличие одного корневого элемента, а также наличие завершающих тегов для каждого начального тега). Если установлена опция UNSAFE, проверка большинства правил формальной корректности не выполняется
xmlDecl([$endoding, $standalone])	В верхнюю часть документа включается XML-объявление. Версия жестко кодируется в виде "1.0"
doctype($name, [$publicId, $systemID])	Добавление объявления типа документа в верхнюю часть документа
comment($text)	Включение XML-комментария
pi($target [, $data])	Вывод обрабатываемой инструкции
startTag($name [, $anamel => $value1, ...])	Создание начального тега элемента. В качестве первого элемента выступает имя элемента, за которым следует пара «имя-значение»
emptyTag($name [, $aname1 => $valuel, ...])	Устанавливает пустой тег элемента. Применяемые в этом случае аргументы будут теми же, что и в случае с элементом startTagO
endTag([$name])	Создает завершающий тег элемента. Не включайте аргумент, если хотите автоматически закрыть текущий открытый элемент
dataElement($name, $data [, $aname => $value1, ...]) $aname => $valuel, ...])	Вывод на печать элемента, содержащего только символьные данные. Этот элемент включает начальный тег, данные и завершающий тег
characters($data)	Вывод части символьных данных

Благодаря использованию перечисленных процедур можно создавать завершенные XML-документы. Например, программа из листинга 3.10 создает базовый HTML-файл.

Листинг 3.10. HTML-генератор

```
use IO;
my $output = new IO::File(">output.xml");
use XML::Writer;
my $writer = new XML::Writer( OUTPUT => $output );
$writer->xmlDecl( 'UTF-8' );
$writer->doctype( 'html' );
$writer->comment( 'My happy little HTML page' );
$writer->pi( 'foo', 'bar' );
$writer->startTag( 'html' );
$writer->startTag( 'body' );
$writer->startTag( 'h1' );
$writer->startTag( 'font', 'color' => 'green' );
$writer->characters( "<Hello World!>" );
$writer->endTag( );
$writer->endTag( );
$writer->dataElement( "p", "Nice to see you." );
$writer->endTag( );
$writer->endTag( );
$writer->end( );
```

В результате выполнения примера выводятся следующие данные:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<!- My happy little HTML page ->
<?foo bar?>
<html><body><h1><font color="green">&lt;Hello World!&gt;</font>
</hl><p>Nice to see you.</p></body></html>
```

В этот модуль встроены небольшие и приятные «удобства». Так, автоматически производится обработка недопустимых символов (например, символам оперсанда, &) путем его преобразования в соответствующие объектные ссылки. Также заключаются в кавычки значения сущностей. В процессе построения документа проверяется содержимое, в котором вы находитесь, с помощью предикативных методов, таких как within_element("foo"). Этот метод сообщает о том, что элемент "foo" является открытым.

По умолчанию модуль отображает документ вместе со всеми активными тегами. В некоторые области XML-кода можно включить пробелы с целью улучшения его восприятия. Если опции NEWLINES присвоить значение true, после тегов элементов будут включаться символы новых строк. Если установить значение опции DATA_MODE, достигается подобный эффект. В результате можно комбинировать значения опций DATA_MODE и DATA_INDENT с целью формирования автоматических отступов строк. Как следствие, обеспечивается генерирование красиво оформленного документа.

Благодаря наличию подобных замечательных свойств возможно применение кода с целью структурирования любого вида текстовых данных. Модуль XML: :Writer позволяет быстро внедрять большой объем информации в ограниченный правилами формальной корректности документ. Например, можно включить листинг каталогов в иерархическую базу данных (листинг 3.11).

Листинг 3.11. Генератор каталогов

```
use XML::Writer;
my $wr = new XML::Writer( DATA_MODE => 'true', DATA_INDENT => 2 );
&as_xml( shift @ARGV );
$wr->end;

# Рекурсивное отображение каталога в XML-код.
#
sub as_xml {
     my $path = shift;
     return unless( -e $path );

     # Если это каталог, создайте элемент и
     # наполните его деталями.
     if( -d $path ) {
         $wr->startTag( 'directory', name => Spath );

         # Загрузка имен элементов каталога
         # в массив.
         my @contents = ( );
         opendir( DIR, $path );
         while( my $item = readdir( DIR )) {
             next if( $item eq '.' or $item eq '..' );
             push( @contents, $item );
         }
         closedir( DIR );
```

```
            # Рекурсивный поиск по элементам каталога.
            foreach my $item ( @contents ) {
                &as_xml( "$path/$item" );
            }

                  $wr->endTag( 'directory' );

        # Мы воздержимся от вызова всего того, что не является
        # каталогом либо файлом.
        } else {
            $wr->emptyTag( 'file', name => $path );
        }
}
```

Ниже приводится пример выполнения кода на основе выбранного каталога (обратите внимание на использование инструкций DATA_MODE и DATA_INDENT с целью улучшения восприимчивости кода):

```
$ ~/bin/dir /home/eray/xtools/XML-DOM-1.25
<directory name="/home/eray/xtools/XML-DOM-1.25">
  <directory name="/home/eray/xtools/XML-DOM-1.25/t">
    <file name="/home/eray/xtools/XML-DOM-1.25/t/attr.t" />
    <file name="/home/eray/xtools/XML-DOM-1.25/t/minus.t" />
    <file name="/home/eray/xtools/XML-DOM-1.25/t/example.t" />
    <file name="/home/eray/xtools/XML-DOM-1.25/t/print.t" />
    <file name="/home/eray/xtools/XML-DOM-1.25/t/cdata.t" />
    <file name="/home/eray/xtools/XML-DOM-1.25/t/astress.t" />
    <file name="/home/eray/xtools/XML-DOM-1.25/t/modify.t" />
  </directory>
  <file name="/home/eray/xtools/XML-DOM-1.25/DOM.gif" />
  <directory name="/home/eray/xtools/XML-DOM-1.25/samples">
    <file
    name="/home/eray/xtools/XML-DOM-1.25/samples/REC-xml-19980210.xml"
    />
  </directory>
  <file name="/home/eray/xtools/XML-DOM-1.25/MANIFEST" />
  <file name="/home/eray/xtools/XML-DOM-1.25/Makefile.PL" />
  <file name="/home/eray/xtools/XML-DOM-1.25/Changes" />
  <file name="/home/eray/xtools/XML-DOM-1.25/CheckAncestors.pm" />
  <file name="/home/eray/xtools/XML-DOM-1.25/CmpDOM.pm" />
```

Итак, мы завершили последовательное рассмотрение модуля XML::Writer, воспользовавшись рекурсивным подходом. Этот подход также удобно применять при работе с объектами древовидной структуры, когда каждый тип XML-объекта обладает собственным методом «приведения к строке». Этот метод предназначен для осуществления соответствующих вызовов объекта редактора. В любом случае, модуль XML::Writer является весьма гибким и полезным в применении.

Другие методы, применяемые для вывода информации

Ранее уже рассматривалось, что многие модули-анализаторы обладают собственными методами, обеспечивающими включение их содержимого в состав XML-кода. Например, модуль XML::LibXML обеспечивает вызов метода toString(), в качестве аргумента которого выступает весь документ либо произвольный объект элемента в составе документа. Следовательно, более специфичные классы процессора, которые являются подклассами данного или какого-либо другого модуля, часто обеспечивают доступность одного и того же метода для собственных API, а также передают пользовательские вы-

зовы метода базовому объекту анализатора. Обратитесь к документации по процессору с целью выяснения того, поддерживается ли это или другое подобное свойство.

И наконец, последнее, что нужно на этом этапе, — функция print языка Perl. Эта функция «обитает» на уровне, который находится ниже инструментального уровня таких инструментов, как XML::Writer. При условии игнорирования специфичных правил и отношений XML, эта функция обеспечивает улучшенную степень контроля над процессом преобразования структур памяти в текст, выводимый на экран. Если вы собираетесь выполнять особо сложную работу, функция print может облегчить вашу задачу. Некоторые из аспектов применения этой функции рассматриваются в главе 10. Не забывайте заменять «надоедливые» символы < и & соответствующими объектными ссылками (см. табл. 2.1), либо заключайте их в секции CDATA.

Наборы символов и кодировки

Независимо от выбора метода управления выводом программы, необходимо учитывать концепцию кодирования символов — протокол вывода XML-документа, используемый для представления различных символов языка (алфавит, каталог идеограмм либо диакритические знаки). Кодировки символов могут представлять сложнейшую часть XML-кода, особенно для программистов из стран Западной Европы и Америки, большинство из которых не знакомы с миром возможных кодировок, отличных от 128-символьной ASCII-кодировки.

Объявление encoding XML-документа может содержать имя произвольной схемы кодировки текста. В соответствии со спецификацией, XML-процессоры допускают применение кодировок UTF-8 и UTF-16. Кодировки UTF-8 и UTF-16 — подмножество Unicode — современной и мощной архитектуры символьной кодировки, позволяющей удовлетворять все потребности программистов.

В этом разделе мы прервем рассмотрение интеграции Perl и XML и перейдем к рассмотрению Unicode. Хотя все примеры, описанные в книге, могут выполняться с применением различного рода наследственных кодировок, все же лучше перейти к использованию современного стандарта — Unicode.

Unicode, Perl и XML

Кодировка Unicode является своего рода «символом цифровой эры», обеспечивающим унификацию тысяч различных систем письменности, разработанных лучшими умами человечества на протяжении столетий. Конечно, если программа выполняется в среде, использующей отличные от ASCII символы, потребуется предварительное знакомство с НИМИ. В любом случае большинство программ обработки текста ограничены низкоуровневыми алфавитно-цифровыми символами из набора Latin. Выбор этого набора символов диктуется его распространенностью в Интернете. С появлением Unicode эта тенденция коренным образом изменилась, а Perl и XML укрепили эти изменения.

В любом документе, «пропагандирующем» достоинства документов Unicode[1], утверждается о том, что Unicode превосходно подходит для выполнения интерна-

[1] Для получения доступа к этим документам воспользуйтесь разделом FAQ, поддерживаемым консорциумом Unicode на web-узле http://unicode.org/unicode/faq/.

ционализации кода. Благодаря его применению программисты могут выполнять локализацию программ, не заботясь об адаптации различных символьных архитектур.

Однако степень важности кодировки Unicode возрастает на порядок в случае возникновения потребности в представлении данных. Языки программирования, применяемые при составлении программ, могут отличаться, но еще большим разнообразием обладают языки, на которых общаются между собой люди. Благодаря использованию Unicode можно создавать единое представление для всех данных, независимо от применяемых языков либо архитектуры.

Кодировки Unicode

Проявляйте осторожность со словами «архитектура» и «кодировка», поскольку Unicode фактически представляет одну из предшествующих кодировок, содержащих истоки всех последующих кодировок.

В Unicode каждый дискретный символ, имеющий официальное толкование, от А до а и ©, имеет *собственный код* — уникальное положительное число, являющееся адресом «карты» Unicode. Например, первой заглавной букве латинского алфавита соответствует шестнадцатеричный адрес 0x0041 (как и в случае с кодами ASCII либо его родственниками). Другим двум символам, строчной греческой букве альфа и символу «смайлика», соответствуют шестнадцатеричные значения, 0x03B1 и 0x263A, соответственно. На основе этих кодов либо их комбинации может конструироваться любой символ. Многие коды символов выделены для хранения диакритических знаков, таких как знаки ударения либо радикалы. В результате многие сценарии представляют собой объединение основных алфавитно-цифровых либо идеографических символов.

Существует великое множество кодировок Unicode. Основное различие между ними заключается в способе представления кодов символов.

Официально Unicode поддерживает три типа кодировки, носящие общее имя UTF (сокращение от слов Unicode Transformation Format), за которым следует число, представляющее наименьший размер (в битах) для представляемых в этой кодировке символов. Имена кодировок будут следующие: UTF-8, UTF-16 и UTF-32. Кодировка UTF-8 является наиболее гибкой и применяется при создании Perl-сценариев.

Кодировка UTF-8

Стиль кодировки UTF-8 является наиболее приближенным к стилю Perl. Она также является наиболее эффективной, поскольку для хранения символов используются отдельные байты. Исходя из этого, UTF-8 по умолчанию применяется для кодирования XML-документов: если в объявлении XML-документа не указан тип кодировки, предполагается использование кодировки UTF-8.

Каждый символ в составе документа, представленный с помощью кодировки UTF-8, использует столько байтов, сколько определяется кодом этого символа (максимум до шести байтов). Таким образом, заглавная литера А представляется с помощью адреса 0x41, требующего один байт памяти. Символу «смайлика» соответствует адрес 0x263A. В этом случае требуются три байта для представления кода символа и один байт — для информирования текстовых процессоров о кодировании символа с помощью нескольких байтов. Вполне вероятно, что в далеком будущем, после вхождения Земли в Галактический союз, для кодирова-

ния символов многочисленных инопланетных языков будут применяться от четырех до шести байтов.

Кодировка UTF-16

Кодировка UTF-16 использует полные два байта для представления символов. Причем характер представления не изменяется даже в том случае, если размер символа позволяет воспользоваться одним байтом (с этой задачей превосходно справляется кодировка UTF-8). С другой стороны, если символ является достаточно редким, для его представления могут потребоваться дополнительные два байта (суррогатная пара). В этом случае общая длина символа составит четыре байта.

ПРИМЕЧАНИЕ

Поскольку в Unicode 2.0 используется стиль представления, соответствующий 16 битам на один символ, в качестве собственной поддерживаемой кодировки, многие говорят о «кодировке Unicode», имея в виду кодировку Unicode UTF-16. Диалоговые окна новых приложений, «Save As...», иногда предлагают отдельные варианты выбора «Unicode» и «UTF-8», даже если эти названия не имеют особого смысла в терминологии Unicode 3.2.

Кодировка UTF-32

Кодировка UTF-32 аналогична кодировке UTF-16, но, в отличие от последней, в UTF-32 каждый символ занимает ровно четыре байта. По этой причине данный тип кодировки не слишком распространен, поскольку в большинстве случаев все передаваемые символы будут снабжены ведущими нулями. Это обстоятельство не позволяет данному виду кодировки демонстрировать чудеса эффективности. Благодаря гарантированной ширине символов возможно представление миллионов символов кодировки Unicode. И хотя UTF-32 относится к трем основным кодировкам Unicode, она не распознается XML-анализаторами. Правда, это обстоятельство вряд ли играет какую-либо заметную роль.

Другие типы кодировок

В XML-стандарте определяется 21 имя для наборов символов, предназначенных для использования анализаторами (помимо общеизвестных кодировок, UTF-8 и UTF-16). Эти имена ранжируются от `ISO-8859-1` (ASCII плюс 128 символов, не связанных с латинским алфавитом) до `Shift_JIS` (кодировка Microsoft для иероглифов японского языка). Хотя по существу эти кодировки не являются кодировками Unicode, каждый представляемый ими символ отображается на один либо большее количество кодов Unicode (и наоборот, позволяя переключаться между различными кодировками).

При использовании XML-анализаторов в Perl возможны собственные методы работы с другими кодировками. Некоторые из них требуют приложения дополнительных усилий. Например, анализатор `XML::Parser` в своем исходном виде мало подходит для работы со многими кодировками. Это связано с тем, что базовая библиотека Expat настроена на восприятие кодировок, отличных от кодировок Unicode. К счастью, это положение можно легко исправить, установив модуль `XML::Encoding` Кларка Купера (Clark Cooper). Этот модуль, по сути, подкласс `XML::Parser` и может считывать, а также воспринимать отображения файлов (XML-документы), которые связывают коды символов, указывающие на другие кодировки, с соответствующими адресами Unicode.

Поддержка Unicode со стороны Perl

Как и в случае с XML, связь между Perl и Unicode укрепляется медленно, но неотвратимо[1]. В общем случае, для корректной работы с кодировкой Unicode необходимо использовать Perl версии 5.6 либо выше. Если у вас установлена эта версия Perl, обратитесь к man-странице perlunicode, где содержатся сведения о текущей поддержке Unicode. Даже если в вашем распоряжении окажется более ранняя версия Perl, всегда можно ее обновить. Вы можете воспользоваться некоторыми дополнительными инструментами, описанными в последующих разделах главы, однако работа с Perl и XML означает использование Unicode, поэтому будет сказываться отсутствие соответствующей поддержки.

В настоящее время наиболее современная и стабильная версия Perl, 5.6.1, обеспечивает частичную поддержку Unicode. Директива utf8 обеспечивает использование в Perl кодировки UTF-8 для большинства функций обработки строк. В Perl также позволяется создавать код на основе символов UTF-8, выходя за пределы обычных ASCII-кодов. Эта возможность пришлась по нраву хакерам, мысли которых передаются с помощью символов, не относящихся к латинскому алфавиту.

В Perl версии 5.8 лучше реализована поддержка Unicode, благодаря чему обеспечивается совместное беспроблемное существование UTF-8 и регулярных выражений. В составе дистрибутива версии 5.8 также появился модуль Encode, имеющий отношение к стандартной библиотеке Perl. Благодаря этому модулю Perl-программисты могут преобразовывать стандартные кодировки в кодировки Unicode:

```
use Encode "from_to";
from_to($data, "iso-8859-3", "utf-8"); # преобразование в utf-8
```

И наконец, версия Perl 6 включила в свой состав все наработки, полученные сообществом Perl за последние двенадцать лет. И конечно же, за это время поддержка Unicode улучшилась. Как говорится, нет предела для совершенства.

Преобразование кодировок

Если вы используете версию Perl 5.8 либо более раннюю, переключение между различными кодировками может потребовать некоторых дополнительных усилий. К счастью, в вашем распоряжении имеются некоторые инструментальные средства, позволяющие решить эту проблему.

iconv и Text::Iconv

По сути, iconv является библиотекой и программой в одном лице, доступной в версиях для Windows и Unix (включая Mac OS X). Она обеспечивает «прозрачный» интерфейс, позволяющий выполнять преобразования типов различных документов. Ниже приводится пример использования этой программы в командной строке Unix:

```
$ iconv -f latin1 -t utf8 my_file.txt > my_unicode_file.txt
```

Если дополнительно к iconv загрузить из сети CPAN Perl-модуль `Text::Iconv`, в вашем распоряжении окажутся Perl API, расширяющие возможности стандарт-

[1] В этом месте повествования вас может посетить романтическая метафора, хотя вы, вероятно, понимаете, что многочисленные «амурные связи» Perl сделали его тем, кем он есть на самом деле.

ной библиотеки. При этом обеспечивается возможность быстрой перекодировки дисковых файлов и строк в памяти.

Модуль Unicode::String

Существует более универсальное решение перекодирования, реализованное в форме модуля Unicode: :String. При использовании этого модуля не требуется базовая библиотека языка С. Базовый интерфейс API является настолько простым, насколько простым могут быть все базовые API. Предположим, что на входе модуля оказывается строка. Она просто передается методу конструктора класса, затем возвращается объект, содержащий строку, наравне с методами, позволяющими выполнять различные операции по обработке текста. Листинг 3.12 демонстрирует тестирование модуля.

Листинг 3.12. Тестирование модуля Unicode::String

```
use Unicode::String;
my $string= "This sentence exists in ASCII and UTF-8, but not UTF-16. Darn!\n";
my $u = Unicode::String->new($string);

# Объект $u содержит объект, представляющий строку.
# 16-битовых символов

# Используется перегрузка, поэтому текстовые Perl-операторы
# делают то, что от них ожидают!
$u .= "\n\nOh. hey, it's Unicode all of a sudden. Hooray!!\n"

# Вывод на печать с применением кодировки UTF-16
#(также известно под именем UCS2)
print $u->ucs2;

# Вывод на печать в красивом виде.
print $u->utf8 :
```

Многие методы модуля позволяют уменьшать количество битов, занимаемых символами. Например, метод utf7 обеспечивает исключение одного бита для символов в кодировке UTF-8.

ПРИМЕЧАНИЕ

Иногда создается впечатление, что модуль XML : : Parser «навязывает» окружающим кодировку Unicode. Независимо от определенной кодировки документа, происходит преобразование всех символов, имеющих высшие кодировки Unicode, в кодировку UTF-8. В случае если пользователь запрашивает данные обратно, происходит обратное преобразование кодировки символов. Подобное «тихое» преобразование может привести к неприятным последствиям. Если вы используете XML: : Parser в качестве основы программы обработки данных, подумайте о возможном использовании упомянутых в разделе инструментов преобразования.

Метки порядка байтов

Если в вашем распоряжении оказывается XML-документ, поступивший из неизвестного источника, о кодировке которого ничего не известно, можно воспользоваться *меткой порядка байтов* (byte order mark, BOM). Обычно эта метка находится в заголовке документа. Документы, использующие кодировки Unicode UTF-16 и UTF-32, зависят от завершающей части (при использовании кодировки UTF-8 благодаря особенностям реализации протокола подобного не происходит). Если неизвестно, где находится завершающая часть байта (важный бит, определяющий чтение документа в обычном и зеркальном порядке), содержимое документа искажается, в силу чего он может оказаться неприемлемым для использования программами.

В качестве метки порядка байтов в Unicode определяется специальная точка кода, U+FEFF. Согласно спецификации Unicode, для документов с кодировкой UTF-16 или UTF-32 должна существовать возможность выделения первых двух либо четырех байтов с целью резервирования символа метки[1]. Таким образом, при считывании программой документа, в случае если значения первых двух битов равны 0xFE и 0xFF, соответственно, это говорит о конце кодировки UTF-16. С другой стороны, наличие подобных кодов свидетельствует о малом завершении документа, поскольку в этом случае отсутствует точка кода Unicode, U+FFFE. (Метки BOM большого и малого завершения в кодировке UTF-32 занимают больше битов: 0x00 0x00 0xFE 0xFF и QxFF 0xFE 0x00 0x00, соответственно.)

В XML-спецификации утверждается, что документы, в которых применяется кодировка UTF-16 и UTF-32, должны включать метки BOM. И опять же, на основе спецификации Unicode можно сделать вывод, что «благоразумные и великодушные» создатели документов передают их по сети с меткой порядка байтов. Другими словами, документы передаются с признаком большого завершения, означающим, что они были объявлены в порядке использования с целью передачи данных между компьютерами. И снова, если вы будете великодушны и благоразумны, то всегда сможете передать информацию о порядке документа по сети в том случае, если не будете уверены в наличии подобного признака порядка. Однако если вы даже почувствуете какие-либо сомнения при получении стандартного документа, не обращайте ни на что внимание и просто воспользуйтесь следующим текстом:

```
open XML_FILE, $filename or die "Can't read $filename: $!";
my $bom; # может включать метку порядка байтов

# Считывание первых двух байтов.
read XML_FILE, $bom, 2;

# Выборка соответствующих числовых значений с помощью
# функции Perl, ord().
my $ordl = ord(substr($bom,0,1));
my $ord2 = ord(substr($bom,1,1));

if ($ordl == 0xFE && $ord2 == 0xFF) {
     # Выглядит подобно документу UTF-16 с большим
     # завершением!
# . . . здесь поступаем соответствующим образом ...

} elsif ($ordl == 0xFF && $ord2 == 0xEF) {
     # O, кто-то подшутил и отослал нам документ с кодировкой
     # UTF-16, которому присуще небольшое завершение.
     # Возможно, вам не захочется выполнять перестановку
     # байтов перед началом их обработки.
} else {
     # Не обнаружена метка порядка байтов.
}
```

Этот пример может пригодиться в том случае, если анализатор не может обнаружить признаки XML-кода в документе. Конечно, в первой строке может быть корректное объявление <?xml ... >, но анализатор не в состоянии правильно считывать XML-код.

[1] Кодировка UTF-8 обладает собственной меткой порядка байтов, применяемой с целью идентификации документа с подобной кодировкой. Вследствие этого полетность метки в мире XML не слишком велика.

Потоки событий



Теперь, когда вы уже получили базовые знания по языку XML, а также имеете представление об анализаторах, настало время приступить к рассмотрению одного из двух важнейших стилей, применяемых в процессе XML-обработки. Речь пойдет о потоках событий. В этой главе рассматриваются некоторые примеры, на основе которых демонстрируется базовая теория потоковой обработки. Вы также познакомитесь с принципами функционирования Simple API for XML (SAX).

Работа с потоками

В мире компьютерных наук *потоком* называется последовательность обрабатываемых фрагментов данных. С этой точки зрения файл может рассматриваться в качестве последовательности символов (каждый из которых требует для своего представления один либо большее количество байтов, в зависимости от применяемого метода кодирования). Программа, использующая эти данные, может устанавливать дескриптор файла. При этом создается поток символов, с помощью которого может выбираться считывание фрагментов данных переменного размера. Потоки также могут генерироваться в динамическом режиме, либо с помощью другой программы, приниматься по сети, либо определяться пользователем. Поток представляет собой абстракцию, причем источник данных не коррелирует с определенной целью обработки.

А теперь приведем описание наиболее общих свойств потока:

- поток состоит из последовательности фрагментов данных;
- имеет значение последовательность передаваемых фрагментов;
- не имеет значения источник данных (файл или выходные данные программы).

По сути, XML-потоки являются группами символьных потоков. Каждая группа данных, которая на диалекте анализатора именуется *маркером,* представляет собой набор из одного или большего количества символов. Каждый маркер соответствует типу разметки, например, начальному либо конечному тегу элемента,

строке символьных данных либо инструкции по обработке. Благодаря этому упрощается разбор XML-кода с применением анализаторов, а также минимизируются требуемые ресурсы и время.

Основное отличие между символьными потоками и XML-потоками заключается в содержимом, присущем каждому маркеру; невозможно заполнить поток случайными тегами и данными, а затем ожидать вразумительных результатов от XML-процессора. Например, поток, включающий десять начальных тегов, для которых не указаны завершающие теги, не представляет особой ценности. И конечно, в данном случае не может идти речь о коде XML, отвечающем требованиям формальной корректности. Данные, которые не подчиняются требованиям формальной корректности, будут исключены из рассмотрения. Но ведь основным назначением XML-кода является упаковка данных с применением метода, гарантирующего целостность структуры и меток документа, не так ли?

Упомянутые контекстные правила применимы в отношении к анализаторам, равно как и к процессорам, выполняющим обработку данных. При разработке языка XML ставилась цель облегчения синтаксического разбора, в то время как другие языки разметки в этой ситуации могли требовать предварительного и заключительного просмотра. Например, в языке SGML отсутствует правило, требующее использования конечного тега для непустых элементов. Определение признака завершения элемента может потребовать от анализатора «сложных размышлений». По этой причине возрастает сложность кода, замедляется скорость обработки, а также увеличивается объем используемой памяти.

События и обработчики

Итак, чем же объясняется название *лоток события?* Почему не поток элементов или объектный поток разметки? Причина заключается в том, что XML иерархичен по своей структуре (элементы содержат другие элементы). В результате невозможно выделять отдельные элементы и использовать их в качестве маркеров потока. В документе, отвечающем правилам формальной корректности, все элементы содержатся в одном корневом элементе. Корневой элемент, содержащий весь документ, сам по себе не является потоком. Поэтому трудно вообразить поток, формирующий завершенный элемент в виде маркера (если речь не идет о пустом элементе).

В силу упомянутых причин XML-потоки составляются на основе событий. Любое *событие* представляет собой сигнал, свидетельствующий об изменении состояния документа. Например, если анализатор обнаруживает начальный тег элемента, сообщается, что был открыт другой элемент и изменилось состояние синтаксического разбора. Завершающий тег влияет на состояние, закрывая последний открытый элемент. XML-процессор может отслеживать открытые элементы в структуре стека данных. При этом в спек помещаются сведения о только что открытых элементах и исключаются данные о закрытых элементах. В любой момент синтаксического разбора процессор может оценить глубину своего нахождения в документе путем оценки размера стека.

Хотя анализаторы поддерживают разнообразные события, могут возникать некоторые «накладки». Например, один анализатор может различать начальный тег и пустой элемент, а другой — нет, но все генерируют сигнал о наличии данного

элемента, А теперь рассмотрим подробнее, каким образом анализатор обрабатывает маркеры (листинг 4.1).

Листинг 4.1. Фрагмент XML-кода

```
<recipe>
  <nane>peanut butter and jelly sandwich</name>
  <!-- add picture of sandwich here -->
  <ingredients>
    <ingredient>Gloppy&trade; brand peanut butter</ingredient>
    <ingredient>bread</ingredient>
    <ingredient>jelly</ingredient>
  </ingredients>
  <instructions>
    <step>5pread peanutbutter on one slice of bread.</step>
    <step>Spread jelly on the other slice of bread.</step>
    <step>Put bread slices together, with peanut butter and
  jelly touching.</step>
  </instructions>
</recipe>
```

В результате применения анализатора в последнем примере может быть сгенерирован следующий перечень событий:

- начало документа (это событие характерно для начала документа, но не для отдельного фрагмента текста);
- начальный тег элемента <recipe>;
- начальный тег элемента <пате>;
- фрагмент текста «peanut butter and jelly sandwich»;
- конечный тег элемента <name>;
- комментарий в форме «add picture of sandwich here»;
- начальный тег элемента <ingredients>;
- начальный тег элемента <ingredient>;
- текст «Gloppy»;
- ссылка на сущность trade;
- текст «brand peanut butter»;
- конечный тег для элемента <ingredient> .

... и так далее , пока не наступит завершающее событие — конец документа.

Где-то между процессами распределением потока на маркеры и их обработкой находится уровень, выполняющий функции диспетчера. С его помощью происходит разветвление процесса обработки, в зависимости от типа маркеров. Код, обрабатывающий конкретный тип маркера, называется *обработчиком.* Причем различные обработчики применяются для работы с начальными тегами, символьными данными и т.д. При создании подобного объекта может применяться совокупность операторов if, с помощью которых организуется вызов различных процедур, предназначенных для обработки того либо иного случая. Также обработчик встраивается в анализатор в качестве диспетчера обратного вызова (потоковый режим XML::Parser). Если зарегистрировать набор процедур в соответствии с типами событий, анализатор вызывает соответствующую процедуру для каждого генерируемого маркера. Применяемая стратегия зависит от выбранного анализатора.

Анализаторы как средство для достижения цели

Нет особой нужды разрабатывать программу обработки XML-кода, выполняющую отделение анализатора от обработчика, хотя подобное действие сулит своп преимущества. Создание модульной программы облегчит организацию и тестирование кода. Идеальный вариант реализуется в том случае, если объекты взаимодействуют друг с другом по выделенным каналам, освобождая их по мере необходимости. Благодаря внедрению принципов модульности облегчается переключение между разделами программы, что является жизненно важным при обработке XML-кода.

Ранее уже упоминалось о том, что XML-поток является абстракцией, не требующей указания источника данных. Все это напоминает своего рода кран во дворе, к которому может подсоединяться поливочный шланг. Причем не имеет значения способ подключения шланга, так как в любом случае основным является требование подачи воды. Также не играет никакой роли конструкция крана и самого шланга. Подобно этому, XML-апализаторы могут рассматриваться как средство для достижения цели: инструмент, который можно загрузить, подключить и наслаждаться его предсказуемым поведением. Сам процесс подключения не составляет особого труда, хотя и может потребовать некоторой изобретательности.

Продолжая аналогию с оборудованием для полива, можно заметить, что ключевым элементом крана является вентиль. Он зависит от величины напора воды и размера поливочного шланга. При работе с XML-потоками событий также требуется поддержка стандартного интерфейса. С этой целью несколько лет назад XML-разработчики перешли на использование SAX. А до недавнего времени XML-анализаторы, реализованные в виде Perl-модулей, не были взаимозаменяемыми. Каждый обладал своим собственным интерфейсом, что приводило к затруднению взаимодействия между ними. Ситуация изменилась после того, как разработчики адаптировали SAX и приняли ряд соглашений, направленных на достижение интеграции между обработчиками и анализаторами. Более подробно эти вопросы будут рассматриваться в главе 5.

Потоковые приложения

Потоковая обработка великолепно подходит при выполнении многих XML-задач. Ниже перечислены некоторые из них:

- фильтр — в результате применения *фильтра* образуется практически идентичная копия исходного документа, содержащая незначительные изменения. Так, например, каждый элемент <A> может преобразовываться в элемент <B>. Обработчик устроен весьма просто и выводит получаемую информацию, выполняя при этом небольшие изменения при обнаружении специфического события;
- селектор — если требуется выбрать часть информации документа, не вдаваясь в подробности остального содержимого, можно воспользоваться *селектором*. В ходе выполнения этой программы «просеиваются» события, осуществляется поиск элемента либо атрибута, содержащего уникальную область данных, именуемую *ключом*. Затем выполнение программы завершается и найденные записи (после завершения повторного форматирования) выводятся на экран;

- сумматор — этот тип программы применяется для создания кратких сводок на основе исходного документа. Например, бухгалтерская программа может рассчитывать заключительный баланс на основе многих записей транзакции: в частности, создается содержание путем вывода названий разделов; генератор индексов может образовывать перечень ссылок на некоторые ключевые слова, выделенные в тексте. Обработчик подобного типа программ должен запоминать части документа с целью его повторной упаковки, после того как анализатор завершит считывание файла;
- преобразователь — тот сложный тип программы предназначен для преобразования одного допустимого формата XML-документа в другой. Например, так можно выполнять преобразование DocBook XML в код HTML. Подобный тип обработки является «венцом» потоковой обработки.

Обработка XML-потоков хорошо подходит для обширного класса задач, хотя следует учитывать некоторые ограничения. Наибольшие проблемы могут возникнуть из-за того, что выполнение всех действий контролируется анализатором, а он может иметь «свое мнение» по разным вопросам. Часто программе требуется воссоздать хронологию событий. Это может выражаться следующим образом: «постой, мне требуется маркер, который был передан 10 шагов назад» либо «не можете ли вы предоставить маркер, находящийся в нижней строке в 12-ти шагах отсюда?». Исследование прошлого возможно в том случае, если предоставить программе дополнительный объем памяти. Сохранение в памяти недавних событий возможно путем грамотного использования структур данных. Однако если потребуется «заглянуть» в далекое прошлое либо в близкое будущее, лучше воспользоваться другой стратегией: обработкой деревьев, рассматриваемой в главе 6.

Ну а теперь, когда вы вооружены базовыми знаниями по обработке XML-потоков, настало время перейти к рассмотрению специфических примеров, связанных с применением на практике XML-потоков.

Анализатор XML::PYX

Во вселенной Perl возникновение стандартных API чрезвычайно замедлено в силу ряда причин. Сеть CPAN, включающая общедоступные модули, растет опережающими темпами. Однако в этой сети отсутствует централизованное руководство, призванное одобрять новые стандарты. Как и в случае с XML, который является новым «игроком» па сцене форматов данных, сообщество пользователей Perl только приступило к разработке своих собственных стандартов.

Новая эра обработки XML-кода средствами Perl может характеризоваться наличием нестандартных анализаторов. В то время модули посиди экспериментальный характер, а документация, как правило, отсутствовала. Это давало простор фантазии и изобретательству, но также и порождало вполне законные опасения. Совершенно неожиданно многие появившиеся в то время инструменты получили весьма широкое распространение. Эта эпоха еще ждет своих исследователей, а мы перейдем к изложению более конкретных вопросов.

Анализатор XML : : PYX является представителем первого поколения анализаторов. Потоки самоорганизуются в соответствии с применяемой концепцией каналов, когда данные, выводимые одной программой, могут направляться другой программе, образуя цепочку процессоров. На основе этой концепции формируется элегант-

ный стиль обработки, который вполне может реализовываться в XML. Здесь важно то, что XML-код может быть реорганизован в виде потока легко распознаваемых коммутативных символов, даже с помощью утилит командной строки.

В нашем примере переформатирования текста с помощью PYX используется символическое кодирование XML-разметки, которое является естественным для таких языков обработки текста, как Perl. Представление каждого XML-события в отдельной строке представляется весьма целесообразным, поскольку многие Unix-утилиты (например, awk и grep) ориентированы на обработку строк. Благодаря этому обеспечивается совместимость PYX с указанными программами (а также с Perl).

В табл. 4.1 приводятся сведения по элементам PYX-записи.

Таблица 4.1 PYX-запись

Символ	Представляет
(	Начальный тег элемента
)	Завершающий тег элемента
-	Символьные данные
A	Атрибут
?	Инструкция по обработке

При наличии какого-либо события в потоке PYX-анализатор открывает новую строку, в начале которой находится один из символов, перечисленных в табл. 4.1. Затем в строке указывается имя элемента или другие подходящие данные. Специальные символы указываются с помощью обратного слэша, как и в любом Perl-сценарии.

Ниже приводится пример преобразования XML-документа в формат PYX-записи, выполняемого анализатором. Следующий XML-код подается на вход анализатора:

```
<shoppinglist>
    <!-- Производитель не играет особой роли -->
    <item>toothpaste</item>
    <item>rocket engine</item>
    <item optional="yes">caviar</item>
</shoppinglist>
```

PYX-запись выглядит следующим образом:

```
(shoppinglist
-\n
(item
-toothpaste
)item
-\n
(item
-rocket engine
)item
-\n
(item
Aoptional yes
-caviar
)item
-\n
)shoppinglist
```

Обратите внимание, что в процессе PYX-трансляции комментарий опускается. Таким образом, PYX несколько упрощает код, исключая некоторые детали раз-

метки. Несмотря на то, что разрешается прогон содержимого, в процессе трансляции пользователю не сообщается о наличии секций разметки CDATA. Наиболее неприятным является то, что из потока исчезают символьные объектные ссылки. Поэтому прежде чем приступить к работе с PYX, необходимо убедиться в том, что вы не нуждаетесь в информации подобного рода.

Мэтт Сержант (Matt Sergeant) разработал модуль, XML : : PYX, выполняющий разбор кода XML, а также его трансляцию в PYX-запись. Небольшая программа выполняет исключение всех тегов XML-элементов, оставляя только символьные данные.

Листинг 4.2. PYX-анализатор

```
use XML::PYX:

# Инициализация анализатора и генерирование PYX-кода.
my Sparser = XML::PYX::Parser->new;
my $pyx;
if (defined ( $ARGV[0] )) {
  $pyx = $parser->parsefile( $ARGV[0] );
}
# Фильтрация тегов.
foreach( split ( / /, $pyx )) {
  print $' if( /^-/ ) ;
}
```

Итак, PYX-анализатор, быстро выполняющий черновую обработку XML-кода, представляет собой неплохую альтернативу SAX и DOM. Его применение ограничивается простыми задачами, например, подсчет количества элементов, отделение содержимого от разметки, а также сообщение о простых событиях. Однако этот анализатор недостаточно «интеллектуален», посему не может применяться для выполнения сложной обработки.

Анализатор XML::Parser

Первый быстрый и эффективный анализатор, ставший фаворитом сети CPAN, назывался XML: :Parser (изощренный интерфейс этого анализатора был рассмотрен в главе 3.). Приступим к знакомству со встроенным потоковым режимом.

Анализатор XML : : Parser будет применяться для считывания списка записей, кодированных в виде XML-документа. Записи содержат персональную контактную информацию, включая имена, адреса и телефонные номера. По мере считывания файла анализатором происходит сохранение информации обработчиком во внутренней структуре данных. После завершения работы анализатора программа сортирует записи по именам и выводит их в виде HTML-таблицы.

Исходный документ приводится в листинге 4.3. Здесь используется корневой элемент <list>, а также четыре вложенных элемента <entry> (определяющие адрес, имя и телефонный номер).

Листинг 4.3. Файл адресной книги

```
<list>
  <entry>
    <name><first>Thadeus</fi rst><last>Wrigley</last></name>
    <phone>716-505-9910</phone>
```

```
      <address>
        <street>105 Marsupial Court</street>
        <city>Fairport</city><state>NY</state><zip>14450</zip>
      </address>
    </entry>
    <entry>
      <name><first>Jill</first><last>Baxter</last></name>
      <address>
        <street>818 S. Rengstorff Avenue</street>
        <zip>94040</zip>
        <city>Mountainview</city><state>CA</state>
      </address>
      <phone>217-302-5455</phone>
    </entry>
    <entry>
      <name><last>Riccardo</last>
      <first>Preston</first></name>
      <address>
        <street>707 Foobah Drive</street>
        <city>Mudhut</city><state>OR</state><zip>32777</zip>
      </address>
      <phone>111-222-333</phone>
    </entry>
    <entry>
      <address>
        <street>10 Jiminy Lane</street>
        <city>Scrapheep</city><state>PA</state><zip>99001</zip>
      </address>
      <name><first>Benn</first><last>Salter</last></name>
      <phone>611-328-7578</phone>
    </entry>
  </list>
```

В этой простой структуре обработка потоков реализуется естественным образом. Каждый начальный тег <entry> определяет подготовку новой части структуры данных для хранения информации. Завершающий тег </entry> свидетельствует о том, что все данные записи собраны и могут быть сохранены. Начальный и завершающий теги подэлементов <entry> указывают обработчику на то, когда и где следует сохранять информацию. Каждый элемент <entry> является самодостаточным (отсутствуют внешние ссылки, благодаря чему облегчается процесс обработки).

Текст программы приводится в листинге 4.4. Верхняя часть кода используется для инициализации объекта анализатора с помощью установки ссылок на процедуры, каждая из которых является обработчиком отдельного события. Подобный стиль обработки событий получил название *обратный вызов*. Этимология этого названия связана с тем, что сначала разрабатывается процедура, а затем осуществляется ее повторный вызов анализатором в случае необходимости обработки событий.

После фазы инициализации осуществляется объявление некоторых глобальных переменных, используемых для хранения информации относительно XML-элементов. Благодаря использованию подобных переменных в распоряжении обработчиков оказывается необходимый объем памяти. Процесс сохранения информации с целью ее последующей выборки часто называют *сохранением состояния*. Применение подобного названия объясняется тем, что в этом случае обработчики могут сохранять состояние синтаксического разбора вплоть до текущей позиции в документе.

После завершения считывания данных и их передачи анализатору в оставшейся части программы определяются процедуры обработчика. Допускается обработка пяти событий: начало и конец документа, начало и конец элементов, а также символьные данные. Другие события, такие как комментарии, инструкции по обработке и объявления типов документа, будут игнорироваться.

Листинг 4.4. Код программы адресной книги

```
# Инициализация анализатора с помощью ссылок на
# процедуры обработчика.
use XML::Parser;
my Sparser = XML::Parser->new( Handlers => {
    Init =>    \&handle_doc_start,
    Final =>   \&handle_doc_end,
    Start =>   \&handle_elem_start,
    End =>     \&handle_elem_end,
    Char =>    \&handle_char_data,
}):
#
# Глобальные переменные.
#
my $record; # Указывает на хэш содержимого элементов.
my $context;# Имя текущего элемента.
my %records;# Набор записей в адресной книге.

#
# Считывание данных и передача их анализатору.
#
my $file = shift @ARGV;
if( $file ) {
    $parser->parsefile( $file );
} else {
    my $input = "";
    while( <STDIN> ) { $input .= $_; }
    $parser->parse( $input );
}
exit;

###
### Обработчики.
###

#
# После начала обработки выводит начало HTML-файла.
#
sub handle_doc_start {
    print "<html><head><title>addresses</title></head>\n";
    print "<body><hl>addresses</hl>\n";
}
#
# Сохранение имени и атрибутов элемента.
#
sub handle_elem_start {
    my( $expat, $name, %atts ) = @_;
    $context = $name;
    $record = {} if( Sname eq 'entry' );
}
# Сбор символьных данных в буфер последних элементов.
```

```
#
sub handle_char_data {
    my( $expat. $text ) = @_;

    # Именование некоторых "капризных" символов.
    $text =~ s/&/&/g;
    $text =~ s/</&lt;/g;

    $record->{ $context } .= $text;
}

#
# Если находится в элементе <entry>,
# направить все данные в запись.
sub handle_elem_end {
    my( $expat, $name ) = @_;
    return unless( $name eq 'entry' );
    my $fullname = $record->{'last'} . $record->{'first'};
    $records{ $fullname } = $record;
}

#
# Вывод, закрытие файла в конце процесса обработки.
#
sub handle_doc_end {
    print "<table border='1'>\n";
    print "<tr><th>name</th><th>phone</th><th>address</th></tr>\n";
    foreach my $key ( sort( keys( %records ))) {
        print "<tr><td>" . $records{ $key }->{ 'first' } . ' ';
        print $records{ $key }->{ 'last' } . "</td><td>";
        print $records{ $key }->{ 'phone' } . "</td><td>";
        print $records{ $key }->{ 'street' } . ' , ';
        print $records{ $key }->{ 'city' } . ' , ';
        print $records{ $key }->{ 'state' } . ' ';
        print $records{ $key }->{ 'zip' > . "</td></tr>\n";
    }
    print "</table>\n</div>\n</body></html>\n";
}
```

Для получения представления о работе программы потребуется ознакомиться с обработчиками. Всем обработчикам, вызываемым модулем XML : : Parser, передается ссылка на объект анализатора expat. Ссылка передается в качестве первого аргумента, благодаря чему облегчается доступ к данным со стороны разработчиков (например, с целью проверки номера строки входного файла). В зависимости от типа события могут передаваться другие аргументы. Например, обработчик события, связанного с начальным элементом, в качестве своего второго аргумента получает имя элемента, а затем получает перечень имен и значений атрибутов.

Для хранения информации обработчики применяют глобальные переменные. Если вы не испытываете симпатий к глобальным переменным (к больших программах их применение может привести к трудностям при отладке), можно создать объект, предназначенный для внутреннего хранения информации. Затем можно передавать анализатору методы объекта, применяемые в качестве обработчиков. В дальнейшем мы будем использовать глобальные переменные, поскольку они лучше воспринимаются в наших примерах.

Первый обработчик носит название handle_doc_start и вызывается в самом начале операции синтаксического разбора. В этом случае обеспечивается удобный

способ для выполнения некоторых подготовительных действий перед началом обработки документа. В рассматриваемом случае осуществляется вывод HTML-кода, который образует HTML-страницу, обеспечивающую форматирование отсортированных записей адресов. Эта процедура не использует какие-либо специальные аргументы.

Следующий обработчик, handle_elem_start, вызывается в том случае, если анализатор обнаруживает начало нового элемента. Кроме обязательной ссылки expat, процедура получает два аргумента: $name, представляющий собой имя элемента, и %atts — хэш имен и значений атрибута. (Обратите внимание на то, что хэш не сохраняет порядок следования атрибутов. Если же этот порядок важен, воспользуйтесь массивом @atts.) В нашем простом примере атрибуты не применяются, однако у вас остается возможность воспользоваться ими в дальнейшем.

Обработка элемента в процедуре обеспечивается путем сохранения его имени в переменной, именуемой $context. Благодаря этому обеспечивается обработка событий, связанных с символьными данными, которые в дальнейшем отсылаются анализатором. Процедура также инициализирует хэш %record, содержащий данные для каждого подэлемента <entry>, оформленные в виде удобной для просмотра таблицы поиска.

Обработчик handle_char_data имеет дело с данными, не относящимися к разметке, на которых основаны все символьные данные, хранящиеся в элементах. Этот текст определяется во втором аргументе, $text. Обработчику требуется лишь сохранить содержимое в буфере, $record->{ Scontext }. Обратите внимание, что вместо назначения символьных данных происходит их передача в буфер. Модуль XML: :Parser обладает интересной особенностью, заключающейся в вызове обработчика символов после разбора каждой строки либо части строки, отделенной символом новой строки[1]. В результате, если содержимое элемента включает символ новой строки, генерируются два отдельных вызова обработчика. Если данные не добавлялись, последний вызов перекрывает предыдущий.

Неудивительно, что обработчики handle_elem_end имеют дело с событиями, связанными с завершением элемента. Имя элемента передается во втором аргументе (как и в случае с обработчиком события начала элемента). Для большинства элементов в этом случае ничего не происходит, но при наличии элемента <entry> придется все же выполнить некоторую работу. На текущий момент времени была собрана вся информация, имеющая отношение к записи. Остается сохранить ее в хэше, проиндексировав в соответствии с именами владельцев данных. Индексация облегчает сортировку записей в дальнейшем. Сортировка может выполняться только после ввода всех записей (поэтому и требуется сохранение записей с целью их дальнейшей обработки). Если же сортировка не требуется, можно просто выводить записи в формате HTML-кода.

И завершает наш набор обработчик handle_doc_end, предназначенный для выполнения заключительных заданий, возникающих после окончания считывания документа. Этот обработчик может потребоваться, например, для вывода па печать записей, отсортированных по именам в алфавитном порядке. Рассматри-

[1] Подобный способ считывания текста присущ только Perl. Поборники чистоты XML могут быть озадачены подобным методом обработки символьных данных. В XM L все служебные символы (и том числе и символы новых строк) трактуются и качестве входных данных.

ваемая процедура генерирует HTML-таблицу, выполняющую форматирование записей.

В рассматриваемом примере приводится простейшая последовательность записей, которая является достаточно простой, однако код XML вовсе не так просто устроен. В некоторых сложных форматах документов приходится учитывать «родителей», «дедушек» или даже «дальних предков» текущего элемента при определении порядка обработки элемента. В этом случае требуется более сложная структура, реализующая сохранение состояния, которая будет рассмотрена в дальнейшем.

SAX



Модуль XML: : Parser реализует функции универсального синтаксического XML-анализатора и генератора потоков, однако у него отсутствуют перспективы в мнре Perl и XML. Проблема заключается в том, что зачастую не требуется анализатор «на все случаи жизни». Просто требуется обеспечить выбор среди нескольких анализаторов, каждый из которых служит достижению некоторой специфической цели. В распоряжении программиста могут оказаться модули, имеющие самый различный характер. Однако не следует ожидать, что применяемый анализатор будет носить универсальный характер. На это не способен даже модуль XML: : Parser, располагающий внушительным набором опций и режимов работы. Поэтому следует создавать разнообразные анализаторы, предусмотренные для применения в самых различных ситуациях.

Среда, включающая различные анализаторы, требует установки некоторого уровня совместимости. Если бы каждый анализатор располагал собственным интерфейсом, разработчикам пришлось бы непросто. Значительно проще освоить один интерфейс, а затем внедрить его поддержку среди различных анализаторов. Практические потребности диктуют создание унифицированного интерфейса между анализатором и программой, — средства, которое является достаточно гибким и надежным, свободным от индивидуальных особенностей конкретного анализатора.

Среда разработки XML основана на управляемом событиями интерфейсе, называемом SAX. Возникновению интерфейса SAX предшествовали дискуссии в списке рассылке XML-DEV. Благодаря поддержке Дэвида Мегинсона (David Megginson)[1] довольно быстро была разработана спецификация этого интерфейса. Первая версия SAX уровня 1 (или просто SAX1), поддерживает элементы, атрибуты и команды обработки. Здесь не предусмотрена обработка некоторых других объектов, например, пространств имен или секций CDATA. Поэтому появилась вторая версия, SAX2, воплощающая любое событие, которое можно представить в общем XML-коде.

[1] Давид Мегинсон поддерживает web-страницу, посвященную SAX, http://www.saxproject.org.

Рост популярности интерфейса SAX происходил очень быстро. Он прост и легок в изучении. Развитие языка XML на ранних стадиях происходило в среде Java, поэтому SAX получил наименование структурного компонента интерфейса. В этом случае подразумевался класс особого вида, в котором методы объекта объявлялись без их реализации (сие действие «ложилось на плечи» разработчика).

Вскоре идеи SAX пронизали все сообщество пользователей Perl. Итогом этого явилось возникновение различных реализаций этого интерфейса в сети CPAN, что, в свою очередь, привело к появлению некоторых проблем. Язык Perl не дает точного способа определения типового интерфейса, как это позволяет делать Java. Ему присущ недостаточный уровень контроля типов и возможной несовместимости. Язык Java сравнивает типы аргументов функции с теми, которые определены в структурном компоненте интерфейса во время компиляции, а Perl просто принимает любые используемые аргументы. Поэтому определение стандартов возложено «на плечи» разработчика и полностью зависит от его опыта и бдительности.

Одна из первых реализаций интерфейса SAX па Perl — это модуль XML: : Parser: : PerlSAX Кена Маклеода (Ken McLeod). Выступая в подкласс XML : : Parser, он изменяет поток событий из Expat, реализуя SAX-события.

Обработчики SAX-событий

При использовании в программе типового SAX-модуля требуется передать ему объект, методы которого реализуют обработчики для SAX-событий. В табл. 5.1 описаны методы типичного объекта обработчика. SAX-анализатор передает хэш-данные каждому обработчику, обладающему соответствующему событию свойствами. Например, в составе этих хэш-данных обработчик элемента получит имя элемента и список его атрибутов.

Таблица 5.1 Обработчики PerlSAX

Имя метода	Событие	Свойства
start_document	Начата обработка документа (первое событие)	(не определены)
end_document	Завершена обработка документа (последнее событие)	(не определены)
start_element	Обнаружен начальный тег элемента или пустой тег элемента	Name, Attributes
end_element	Обнаружен конечный или пустой тег, соответствующий элементу	Name
characters	Обнаружена строка символов, не являющихся символами разметки (символьные данные)	Data
processing_ i instruction	Найдена команда обработки	Target, Data
comment	Встречен комментарий	Data
start_cdata	Обнаружено начало секции CDATA (следующие символьные данные могут содержать зарезервированные символы разметки)	(не определены)
end_cdata	Обнаружен конец секции CDATA	(не определены)
enti ty_reference	Обнаружена внутренняя ссылка объекта (альтернативная внешней ссылке объекта, которая указывает на то, что необходимо загрузить файл)	Name, Value

Несколько замечаний относительно методов обработчика:

- для пустых элементов вызываются обработчики start_element () и end_element(), для пустых элементов отсутствует какой-либо специальный обработчик;
- обработчик characters () может вызываться многократно для строки непрерывных символьных данных, разбивая ее при этом на части (например, анализатор может «разрывать» текст вокруг ссылки объекта, что зачастую имеет определенный смысл);
- при наличии пробела между элементами вызывается обработчик characters () (даже в том случае, если не рассматриваются значимые данные);
- разбор команд обработки, комментариев и секций CDATA необязателен. При отсутствии обработчиков данные, имеющие отношение к командам обработки и комментариям, пропускаются. Для секций CDATA по-прежнему вызывается обработчик characters (), поэтому данные не будут утеряны;
- обработчики start_cdata() и end_cdata() не получают данных. Вместо этого они выполняют функцию сигналов, указывающих н то, можно ли ожидать, что полученные символы разметки приведут к вызову обработчика characters();
- При отсутствии обработчика entity_reference () все внутренние ссылки объекта будут переданы анализатору автоматически, а полученный в результате текст или разметка будут обрабатываться обычным образом. Если определен обработчик entity_reference(), то объектные ссылки не будут расширяться и с ними можно делать все что угодно.

Теперь приведем пример. Напишем программу, вызывающую фильтр, который дублирует исходного документа с некоторыми модификациями. Суть изменений заключается в следующем:

- каждый XML-комменарий превращается в элемент <comment>;
- удаляются команды обработки;
- удаляются теги, но остается содержимое элементов <literal>, которые отображаются в элементах <programli sti ng> на любом уровне.

Код этой программы приведен в листинге 5.1. Как и в предыдущей программе, инициализируется анализатор с набором обработчиков, за исключением того, что па этот раз они объединяются в составе удобного пакета — объекта, получившего название MyHandler. Отметим, что здесь реализовано несколько больше обработчиков, поскольку требуется обработка комментариев, команды обработки и пролог документа.

Листинг 5.1. Программа-фильтр

```
# Инициализация анализатора.
#
use XML::Parser::PerlSAX;
my Sparser = XML::Parser::PerlSAX->new( Handler => MyHandler->new( ) );
if( my $file = shift @ARGV ) {
    $parser->parse ( Source => {SystemId => $file} );
} else {
    my $input = "";
```

```
    while( <STDIN> ) { $input .= $_; }
        $parser->parse( Source => {String => Sinput} );
}
exit;
#
# Глобальные переменные.
#
my @element_stack;  # Запоминайте имена элементов.
my $in_intset; # Флаг: внутреннее подмножество?
###
### Пакет обработчиков документа.
###
package MyHandler;
#
# Инициализация пакета обработчиков.
#
sub new {
    my $type = shift;
    return bless {}, Stype;
}
#
# Обработка события начала элемента:
# Вывод начального тега и атрибутов.
#
sub start_element {
    my( $self, $properties ) = @_;
    # Отметим, что хэш-данные %{$properties} утеряют
    # порядок следования атрибутов.
    # Закрыть внутреннее подмножество в случае, если
    # оно пока открыто.
    output( "]>\n" ) if( $in_intset );
    $in_intset = 0;
    # Запомнить имя, записав его в стек.
    push( @element_stack, $properties->{'Name'} );
    # Вывести тег и атрибуты, кроме <literal>
    # внутри <programlisting>.
    unless( stack_top( 'literal' ) and
        stack_contains( 'programlisting' )) {
        output( "<" . $properties->{'Name'} );
        my %attributes = %{$properties->{'Attributes'}};
        foreach( keys( %attributes )) {
        output( " $_=\"" . $attributes{S_} . "\"" );
        }
        output( ">" );
    }
}
#
# Обработка события конца элемента: отображение конечного
# тега за исключением <literal> внутри <programlisting>.
#
sub end_element {
    my( Sself, Sproperties ) = @_;
    output( "</" . $properties->{'Name'} . ">" )
        unless( stack_top( 'literal' ) and
            stack_contains( 'programlisting' ));
    pop( @element_stack );
}
#
```

```
# Обработать событие, связанное с наличием символьных
# данных.
#
sub characters {
    my( $self, $properties ) = @_;
    # К сожалению, анализатор разбирает некоторые
    # символьные объекты, поэтому необходимо еще раз
    # переместить их вместе со ссылками объектов.
    my Sdata = $properties->{'Data'};
    $data =- s/\&/\&/;
    $data =~ s/</\&lt;/;
    $data =- s/>/\&gt;/;
    output( Sdata );
}
#
# Обработать событие комментария: включение в
# элемент <comment>.
#
sub comment {
    my( Sself, $properties ) = @_;
    output( "<comment>" . $properties->{'Data'} . "</comment>" );
}
#
# Обработать событие PI: удалить его
#
sub processing_i instruction {
    # Ничего не делаем!
}
#
# Обработать ссылку внутреннего объекта
# (его не нужно разбирать) .
#
sub entity_reference {
    my( Sself, $properties ) = @_;
    output( "&" . $properties->{'Name'} . ":" );
}
sub stack_top {
    my $guess = shift;
    return $element_stack[ $#element_stack ] eq $guess;
}
sub stack_contains {
    my $guess = shift;
    foreach( @element_stack ) {
        return 1 if( $_ eq Sguess );
    }
    return 0;
}
sub output {
    my $string = shift;
    print $string;
}
```

Внимательно присмотревшись к обработчикам, можно увидеть, что кроме обязательной ссылки объекта $self, передается один аргумент. Этот аргумент — ссылка на хэш-данные свойств события. Этот метод обладает одним недостатком, который заключается в том, что: в обработчике начала элемента атрибуты хранятся в хэш-данных, причем не соблюдается исходный порядок атрибутов. С позиций

семантики это не имеет значения, так как в XML допускается несоблюдение порядка атрибутов. Однако возможны случаи, когда необходимо воспроизводить этот порядок[1].

При использовании этой программы в качестве фильтра сохраняется все содержимое исходного документа, за исключением нескольких деталей, которые должны измениться. Программа фиксирует пролог документа, команды обработки и комментарии. Должны сохраниться даже ссылки объекта, поскольку они заменяют разбираемые ссылки (так как анализатор может этого потребовать). Поэтому программа включает больше обработчиков, чем в предыдущем примере, в котором нас интересовала лишь выборка весьма специфической информации.

Теперь проверим эту программу в действии. Наш входной файл данных приведен в листинге 5.2.

Листинг 5.2. Данные для программы-фильтра

```
<?xml version="1.0"?>
<!DOCTYPE book
     SYSTEM "/usr/local/prod/sgml/db.dtd"
[
     <!ENTITY thingy "hoo hah blah blah">
]>
<book id="mybook">
<?print newpage?>
     <title>GRXL in a Nutshell</title>
     <chapter id="intro">
          <title>What is GRXL?</title>
<!-- Требуется лучшее название -->
          <para>
Yet another acronym. That was our attitude at first, but then we saw
the amazing uses of this new technology called
<literal>GRXL</literal>. Consider the following program:
          </para>
<?print newpage?>
          <programlisting>AH aof -- %%%%
{{{{{{ let x = 0 }}}}}}
     print! <lineannotation><literal>wow</literal></lineannotation>
or not!</programlisting>
<!-- Какой шрифт нам нужен? -->
     <para>
What does it do? Who cares? It's just lovely to look at. In fact,
I'd have to say, "&thingy;".
          </para>
<?print newpage?>
     </chapter>
</book>
```

[1] В случае применения программы-фильтра может потребоваться сравнить версии исходного и обработанного файла, используя утилиту diff. В результате выполнения подобного сравнения будет выявлено много отличий, причина которых заключается в изменении порядка следования атрибутов. В этом случае следует воспользоваться модулем XML::SemanticDiff Кипа Хэмптона (Kip Hampton). Этот модуль будет игнорировать синтаксические отличия, а также сравнивать только семантику двух документов.

Результат, полученный после запуска на выполнение программы с этими данными, представлен в листинге 5.3.

Листинг 5.3. Результат выполнения программы-фильтра

```
<book id="mybook">
    <title>GRXL in a Nutshell</title>
     <chapter id="intro">
          <title>What is GRXL?</title>
<comment> need a better title </comment>
          <para>
Yet another acronym.  That was our attitude at first, but then we saw
the amazing uses of this new technology called
<literal>GRXL</literal>. Consider the following program:
          </para>
          <programlisting>AH aof -- %%%%
{{{{{{ let x = 0 }}}}}}
     print! <lineannotation>wow</lineannotation>
or not!</programlisting>
<comment> what font should we use? </comment>
          <para>
What does it do?  Who cares?  It's just lovely to look at.  In fact,
I'd have to say, "&thingy;".
          </para>
     </chapter>
</book>
```

А теперь проверим, что было сделано верно программой-фильтром. Она преобразовала XML-комментарий в элемент <comment> и удалила команды обработки. Элемент <literal>, входящий в состав элемента <programlisting>, был удален, причем его содержимое не разбиралось, в то время как другие элементы <literal> сохранились. Объектные ссылки остались неразрешенными, как и требовалось. Пока все правильно. Но кое-чего не хватает. Исчезли XML-объявление, объявление типа документа и внутреннее подмножество. Без объявления объекта thingy этот документ не достоверен. Оказалось, что имеющихся в наличии обработчиков недостаточно.

DTD-обработчики

Модуль XML::Parser::PerlSAX поддерживает другую группу обработчиков, которые используются для обработки DTD-событий. Причем субъектом в этом случае выступает все, что находится до таких основных элементов, как XML-объявлеиие, объявление типа документа и внутреннего подмножества объекта и объявлений объекта. Все это называется *прологом документа.* Если нужно получить документ в исходном виде (например, в программе-фильтре), необходимо определить некоторые из этих обработчиков для порождения пролога документа. Определение этих обработчиков — это именно то, что нужно было выполнить в предыдущем примере.

Подобные обработчики могут применяться в других целях. Например, возможна предварительная загрузка описания объектов с целью специальной обработки, вместо того чтобы надеяться на то, что анализатор сделает эту подстановку по умолчанию. Эти обработчики приведены в табл. 5.2.

Таблица 5.2 Обработчики PerlSAX DTD

Имя метода	**Событие**	**Свойства**
entity_decl	Анализатор воспринимает объявление сущности (внутренней или внешней, прошедшей или не прошедшей синтаксический разбор)	Name, Value, PublicId, SystemId, Notation
notation_decl	Анализатор обнаружил объявление записи	Name, PublicId, SystemId, Base
unparsed_entity_decl	Анализатор обнаружил объявление сущности, не прошедшей синтаксический разбор (например, двоичного элемента данных)	Name, PublicId, SystemId, Base
element_decl	Обнаружено объявление элемента	Name, Model
attlist_decl	Встречено объявление списка атрибутов элемента	ElementName, AttributeName, Type, Fixed
doctype_decl	Анализатор нашел объявление типа документа	Name, SystemId, PublicId, Internal
xml_decl	Найдено XML-объявление	Version, Encoding, Standalone

Обработчик entity_decl () вызывается для всех видов объявлений сущностей, если не определен более специфический обработчик. Поэтому объявления сущностей, не прошедших синтаксический разбор, приводят к запуску обработчика entity_decl(), если не определен обработчик unparsed_entity_decl(), который имеющий более высокий приоритет.

Параметры обработчика entity_decl () изменяются в зависимости от типа сущности. Параметр Value устанавливается для внутренних переменных, но не для внешних. Параметры PublicId и SystemId, которые указывают XML-процессору, где найти файл, содержащий значение объекта, устанавливаются только для внешних переменных и не определяются для внутренних переменных. Параметр Base сообщает процессору о том, что будет использоваться в качестве базовой URL-ссылки в случае, если параметр SystemId содержит относительный адрес.

Объявления записи — это специальное свойство DTD, допускающее назначение специального идентификатора типа для объекта. Например, можно объявить тип данных объекта "date", сообщив XML-процессору о том, что объект нужно рассматривать в качестве объявленного типа данных. Это свойство не часто используется в XML-коде, поэтому в дальнейшем не будем рассматривать подобную ситуацию.

Свойство Model обработчика element_decl () включает модель содержимого или грамматику для элемента. С помощью этого свойства описываются свойства, включаемые внутрь элемента в соответствии с DTD-объявлением.

Объявление типа документа — это необязательная часть заголовка документа, расположенная непосредственно под XML-объявлением. Параметр Name определяет имя корневого элемента в документе. С помощью параметров PublicId и SystemId процессор получает сведения о местонахождении внутреннего подмножества DTD. Параметр Internal определяет все внутреннее подмножество в виде строки на тот случай, если требуется пропустить отдельный объект и не выполнять обработку объявления элемента.

Предположим, что необходимо дополнить код листинга программы-фильтра, реализующего получение пролога документа именно в том виде, в котором он разбирается анализатором. В таком случае необходимо определить обработчики, как описано в листинге 5.4.

Листинг 5.4. Усовершенствованная программа-фильтр

```
# Обработка xml-объявления.
#
sub xml_decl {
     my( $self, $properties ) = @_;
     output( "<?xml version=\"" . $properties->{'Version'} . "\"" );
     my $encoding = $properties->{'Encoding'};
     output( " encoding=\"$encoding\"" ) if( $encoding );
     my $standalone = $properties->{'Standalone'};
     output( " standalone=\"$standalone\"" )
          if( $standalone );
     output( "?>\n" );
}
#
# Обработка объявления типа документа:
# попытаемся скопировать оригинал.
#
sub doctype_decl {
     my( $self, $properties ) = @_;
     output( "\n<!DOCTYPE " . $properties->{'Name'} .
          "\n" );
     my $pubid = $properties->{'PublicId'};
     if( $pubid ) {
          output( "  PUBLIC \"$pubid\"\n" );
          output( "  V" . $properties->
               {'SystemId'} . "\"\n" );
     } else {
          outputC "  SYSTEM V" . $properties->
               {'SystemId'} . "\"\n" );
     }
     my $intset = $properties->{'Internal'};
     if( $intset ) {
          $in_intset = 1;
          output ( " [\n" ) :
     } else {
          output ( ">\n" );
     }
}
#
# Обработать объявление сущности во внутреннем
# подмножестве, восстановить оригинальное объявление
# в прежнем виде.
#
sub entity_decl {
     my( $self, $properties ) = @_;
     my $name = $properties->{'Name'};
     outputC "<!ENTITY $name " );
     my $pubid = $properties->{'PublicId'};
     my $sysid = $properties->{'SystemId'};
     if( $pubid ) {
          output( "PUBLIC \"$pubid\" \"$sysid\"" );
```

```
        } elsif( $sysid ) {
            output( "SYSTEM \"$sysid\"" );
        } else {
            output( "\"" . $properties->{'Value'} . "\"" );
        }
            output( ">\n" ) ;
    }
```

Теперь рассмотрим, как выглядит результат выполнения программы-фильтра (см. листинг 5.5).

Листинг 5.5. Результат выполнения программы-фильтра

```
<?xml version="1.0"?>
<!DOCTYPE book
    SYSTEM "/usr/local/prod/sgml/db.dtd"
[
<!ENTITY thingy "hoo hah blah blah">
]>
<book id="mybook">
    <title>GRXL in a Nutshell</title>
    <chapter id="intro">
        <title>What is GRXL?</title>
<comment> need a better title </comment>
        <para>
Yet another acronym.  That was our attitude at first, but then we saw the
amazing uses of this new technology called
<literal>GRXL</literal>.  Consider the following program:
        </para>
        <programlisting>AH aof -- %%%%
{{{{{{ let x = 0 }}}}}}
    print!  <lineannotation>wow</lineannotation>
    or not!</programlisting>
    <comment> what font should we use? </comment>
        <para>
What does it do? Who cares? It's just lovely to look at.  In fact,
I'd have to say. "&thingy;".
        </para>
    </chapter>
</book>
```

Так гораздо лучше. Создана завершенная программа-фильтр. Основные обработчики занимаются элементами и их содержимым, а DTD-обработчики – всем тем, что может произойти вне корневого элемента.

Разрешение внешних сущностей

По умолчанию анализатор заменяет все объектные ссылки объекта их фактическими значениями. Как правило, это именно то, что нужно, но иногда, как в примере программы-фильтра, предпочтительнее сохранить объектные ссылки. Эту операцию можно выполнить нутем включения метода обработчика enti ty_reference (). Неясным остается вопрос о том, как переопределить обработку по умолчанию ссылок внешних объектов. Также анализатор заменяет ссылки их значениями посредством выяснения местонахождения файлов н включения их содержимого в поток. Потребуется ли в какой-нибудь ситуации изменить это поведение и, если да, как это следует сделать?

Сохранение документов в нескольких файлах удобно, особенно в случае действительно больших документов. Например, в случае наличия большой книги, написанной с применением XML-кода, каждая глава этой книги должна сохраняться в отдельном файле. Это можно выполнить без особого труда посредством применения внешних сущностей. Например:

```
<?xml version="1.0"?>
<doctype book [
     <!ENTITY intro-chapter SYSTEM "chapters/intro.xml">
     <!ENTITY pasta-chapter SYSTEM "chapters/pasta.xml">
     <!ENTITY stirfry-chapter
     SYSTEM "chapters/stirfry.xml">
     <!ENTITY soups-chapter
     SYSTEM "chapters/soups.xml"> ]>
<book>
     <title>The Bonehead Cookbook</title>
     &intro-chapter;
     &pasta-chapter;
     &stirfry-chapter;
     &soups-chapter;
</book>
```

В предыдущем примере программа-фильтр ориентирована на разрешение внешних объектных ссылок, в резул ьтате всего вся книга будет выведена в виде одной сущности. В таком случае схема разбиения файла была бы утеряна, и пришлось бы выполнять редактирование путем его повторного разбиения на несколько частей. К счастью, можно переопределить разрешение внешних объектных ссылок, используя обработчик resolve_entity().

Этот обработчик включает четыре свойства: Name — имя объекта; SystemId и PublicId — идентификаторы, которые позволяют определять местоположение файла, содержащего текст объекта. Также определяется параметр Base, который помогает разобрать соответствующие URL-ссылки, если таковые существуют. В отличие от других подобных случаев этот обработчик должен возвращать значение, которое указывает анализатору на выполняемые действия. Возврат значения undef указывает обработчику на то, что требуется загрузить внешнюю сущность в ее исходном виде. В противном случае необходимо возвратить хэш-данные, описывающие альтернативный источник, с применением которого производится загрузка объекта. Они имеют тот же тип, который используется для передачи сущности с применением метода parse(), причем помощью ключа SystemId передается имя файла или URL, а String — строка текста. Например:

```
sub resolve_entity {
     my( $self, $props ) = @_;
     if( exists( $props->{ SystemId }) and
          open( ENT, $props->{ SystemId })) {
          my $entval = '<?start-file ' . $props->
          { SystemId } . '?>':
          while( <ENT> ) { $entval .= $_; }
          close ENT;
          $entval .= '<?end-file ' . $props->
          { SystemId } . '?>' ;
          return { String => $entval };
     } else {
```

```
            return undef;
        }
    }
```

Эта процедура открывает источник сущности в найденном файле и передает его анализатору в виде строки. Сначала подключается команда обработки, помещенная до и после текста объекта, отмечая границу файла. Затем можно написать подпрограмму, которая производит поиск инструкций по обработке и вновь разбивает файлы.

Драйверы источников, не включающих XML-код

В листинге, описывающем программу-фильтр, использовался файл, содержащий XML-документ в качестве источника входных данных. Этот пример демонстрирует лишь один из многих способов использования SAX. Другое его распространенное применение — чтение данных с применением драйвера, который формирует поток данных из источника, не являющимся XML-кодом, например из базы данных. Драйвер SAX преобразует поток данных в последовательность SAX-событий, которые могут обрабатываться с помощью ранее применяемого способа. Достоинство этого метода состоит в том, что можно использовать одну и ту же программу независимо от имеющегося источника данных. Поток SAX-событий абстрагируется от данных и разметки, поэтому эти объекты игнорируются при дальнейшей обработке. Изменения, вносимые в программу для работы с файлами и другими драйверами, будут незначительными.

С целью отслеживания выполнения драйвера создадим программу, которая использует модуль XML : : SAXDriver: : Excel Ильи Стерина (Ilya Sterin) с целью преобразования электронных таблиц Microsoft Excel в XML-документы. Этот пример иллюстрирует, каким образом реализуется конвейерный способ обработки потока данных. Объект Spreadsheet: :ParseExcel считывает файл и генерирует общий поток данных, который объект XML: : SAXDriver: :Excel преобразует в поток SAX-событий. Затем этот поток выводится программой в качестве XML-документа.

Рассмотрим следующую электронную таблицу в формате Excel:

	A	B
1	Бейсбольный мяч	55
2	Теннисный мяч	33
3	Мяч для настольного тенниса	12
4	Футбольный мяч	77

SAX-драйвер будет порождать новые элементы, передавая имена в виде аргументов с целью вызова методов обработчика. Выведем их па печать в исходном виде, а затем проследим, каким образом драйвер структурирует документ. Описанные действия реализованы кодом, приведенным в листинге 5.6.

Листинг 5.6. Программа разбора таблицы Excel

```
use XML::SAXDriver::Excel;
# Получить имя файла для обработки.
die( "Must specify an input file" ) unless( @ARGV ):
my $f1le = shift @ARGV;
print "Parsing $file...\n";
```

```
# Инициализация анализатора.
my Shandler = new Excel_SAX_Handler;
my %props = ( Source => { SystemId => $file },
          Handler => Shandler );
my $driver = XML::SAXDriver::Excel->new( %props );
# Начать синтаксический разбор.
$driver->parse( %props );
# Определенный нами пакет обработчиков выведет XML-документ
# после получения SAX-сообщений.
package Excel_SAX_Handler;
# Инициализация пакета.
sub new {
     my $type = shift;
     my $self = {@_};
     return bless( Sself, $type );
}
# Создание самого внешнего элемента.
sub start_document {
     print "<doc>\n";
}
# Конец элемента документа,
sub end_document {
     pri nt "</doc>\n";
}
# Обработка любых символьных данных.
sub characters {
     my( Sself, $properties ) = @_;
     my $data = $properties->{'Data'};
     print $data if defined($data);
}
# Начало нового элемента, вывод начального тега.
sub start_element {
     my( Sself, $properties ) = @_;
     my $name = $properties->{'Name'};
     print "<$name>";
}
# Конец нового элемента.
sub end_element {
     my( Sself, $properties ) = @_;
     my $name = $properties->{'Name'};
     pri nt "</$name>";
}
```

Нетрудно заметить, что методы обработчика весьма похожи на те, которые использовались в предыдущем примере с SAX. Изменились лишь действия, которые производятся по отношению к аргументам. Теперь рассмотрим, каким образом выглядит результат, когда в качестве исходных данных программы служит следующий тестовой файл:

```
<doc>
<records>
     <record>
               <column1>baseballs</column1>
               <column2>55</column2>
          </record>
          <record>
               <column1>tennisballs</column1>
               <column2>33</column2>
          </record>
```

```
            <record>
                <column1>pingpong balls</column1>
                <column2>12</column2>
            </record>
            <record>
                <column1>footballs</column1>
                <column2>77</column2>
            </record>
            <record>
Use of uninitialized value in print at conv line 39.
                <column1></column1>
Use of uninitialized value in print at conv line 39.
                <column2></column2>
            </record>
</records></doc>
```

Драйвер выполнил большую часть действий по созданию элементов и форматированию данных. Все, что было выполнено нами, — вывод пакетов, переданных в форме вызовов метода. Весь документ был заключен в теги <records>, благодаря чему использование тега <doc> стало излишним. (При следующей переработке программы сделаем так. чтобы методы start_document() и end_document() ничего не выводили.) Каждая строка электронной таблицы помещается внутри тегов <record>. Наконец, два столбца различаются метками <column1> и <column2>. В итоге получился весьма неплохой результат.

Как видим, с минимальными усилиями были использованы возможности SAX для выполнения довольно сложной работы по преобразованию одного формата в другой. Фактически, драйвер автоматизирует процесс преобразования. При этом достигается уровень гибкости, достаточный для интерпретации событий. Поэтому можно отказаться от применения неподходящих данных (например, пустых строк) или переименовать элементы. Также можно выполнить сложную обработку, например, найти сумму значений и упорядочить строки.

Базовый класс обработчиков

Интерфейс SAX не различает элементы, предоставляя возможность выполнять эти действия разработчикам. Необходимо рассортировывать имена элементов в обработчике start_element() и, возможно, использовать стек, чтобы отслеживать иерархию элементов. Нет ли какого-либо способа обобщения излагаемых идей? Для этого предназначается модуль XML::Handler::Subs Кена МакЛеода (Ken MacLeod).

Здесь определен объект, который группирует вызовы обработчика для более специфических обработчиков. Если нужен обработчик только для элементов <title>, его создание не составит особого труда. Обработчик для начального тега должен начинаться комбинацией s_, за которой следует имя элемента (специальные символы заменяются подчеркиванием). Обработчик конечного тега имеет аналогичный вид, но начинается комбинацией символов e_.

Это еще не все. Базовый объект также имеет встроенный стек и предоставляет метод доступа, применяемый с целью проверки нахождения анализатора внутри определенного элемента. Переменная $self->{Names} ссылается на стек имен элементов. Метод in_element($name) используется в любой момент времени для проверки, находится ли анализатор внутри элемента, имеющего имя $name.

С целью выполнения проверки напишем программу, которая производит специфическую обработку элементов. При анализе файла, включающего HTML-код, программа выводит содержимое элемента `<h1>`, включая встроенные элементы, используемые для визуального выделения (например, выделения с помощью специального шрифта). Программа, код которой представлен в листинге 5.7, чрезвычайно проста.

Листинг 5.7. Программа, реализующая создание подклассов на базе обработчиков

```
use XML::Parser::PerlSAX;
use XML::Handler::Subs

#
# Инициализация обработчика.
#
use XML::Parser::PerlSAX;
my Sparser = XML::Parser::PerlSAX->new( Handler => Hl_grabber->new() );
$parser->parse( Source => {SystemId => shift @ARGV} );
## Объект обработчика: Hl_grabber.
##•
package Hl_grabber;
use base( 'XML::Handler::Subs' );
sub new {
        my $type = shift;
        my $self = {@_};
        return bless( $self, $type );
}
#
# Обработка начала документа.
#
sub start_document {
    SUPER::start_document( );
    print "Summary of file:\n";
}

#
# Обработка начального тега <hl>: использовать скобку в
# качестве ограничителя.
#
sub s_hl {
    print "[";
}

#
# Обработка конечного тега <hl>: использовать
# скобку в качестве ограничителя.
#
sub e_hl {
    print "]\n";
}
#
# Обработка символьных данных.
#
sub characters {
    my( $self, $props ) = @_;
    my $data = $props->{Data};
    print $data if( $self->in_element( h1 ));
}
```

Запустим программу на выполнение, воспользовавшись следующим тестовым файлом:

```
<html>
    <head><title>The Life and Times
    of Fooby</title></head>
    <body>
        <h1>Fooby as a child</h1>
    <p>...</p>
        <h1>Fooby grows up</h1>
        <p>...</p>
        <h1>Fooby is in <em>big</em> trouble!</h1>
        <p>...</p>
    </body>
</html>
```

Результат выполнения программы имеет вид:

```
Summary of file:
[Fooby as a child]
[Fooby grows up]
[Fooby is in big trouble!]
```

Благодаря обращению к методу `in_element()` включается текст элемента `< em >`. Определенно, модуль `XML::Handler::Subs` весьма полезен для выполнения SAX-обработки.

XML::Handler::YAWriter как базовый класс обработчиков

Модуль `XML::Handler::YAWriter` Майкла Коэна (Michael Koehn) – пример «еще одного» генератора XML-кода. Он может регистрировать сам себя, но в то же время является автоматически настраиваемым базовым классом, предназначенным для выполнения всех видов действий, связанных с SAX.

Если вы когда-либо работали с различными базовыми классами Perl вида `Tie::*`, то поймете, что здесь применяется сходная идея: вы начинаете с базового класса с определенными обратными вызовами, которые способствуют выполнению всех вызовов подпрограмм, запущенных SAX-событиями. В своем классе драйверов достаточно переопределить подпрограммы, которые должны выполнять какие-либо специальные действия и воспользоваться правилами поведения, заданными по умолчанию для всех событий, обработка которых необязательна.

В этом случае поведение, заданное по умолчанию, также обеспечивает некоторые преимущества: доступ к массиву строк (хранящемуся в виде переменной экземпляра в объекте обработчика) блокирует XML-документ, который создает входящие SAX-события. Это не так важно, если источником данных был XML-код, но если для формирования потока событий из случайного источника данных используется драйвер в стиле PerlSAX, эта особенность будет весьма полезной. Она обеспечивает простой способ, например, преобразования файла, не являющегося XML-документом, в его XML-эквивалент с последующим сохранением копии на диске.

Применяемый в этом случае компромисс заключается в том, что не стоит забывать выполнять `$self->SUPER::[имя_метода]` при реализации методов обработки событий. В противном случае ваш класс может «потерять свои корни» и не вклю-

чит элементы во внутренний массив строк в его исходном представлении, из-за чего в полученном XML-документе появятся «дыры».

Второе поколение XML::SAX

При распространении SAX-анализаторов возникают две проблемы. Сложности проявляются в ходе их приведения в соответствие со стандартными API, а также в ходе организации в своей собственной системе. Созданный совместными усилиями Мэтта Сержанта (Matt Sergeant), Кипа Хэмптона (Kip Hampton) и Робина Берджена (Robin Berjon) модуль XML : : SAX позволяет решать эти проблемы одновременно. Также обеспечивается поддержка для SAX уровня 2, отсутствующая в предыдущих модулях.

Спрашивается: «В чем суть поддержки синхронизации модулей и API?». До сих пор мы предлагали использовать такой стандарт, как SAX, для обеспечения взаимозаменяемости модулей. Однако при этом проявляется свой недостаток: в Perl существует несколько способов реализации SAX. Интерфейс SAX первоначально предназначался для языка Java, в котором присутствует замечательный тип интерфейса класса. С помощью этого типа можно фиксировать, например, тип аргумента, передаваемый конкретному методу. Ничего подобного в Perl нет.

В ранних SAX-модулях, как упоминалось ранее, в этом случае не возникало особых проблем. Все они поддерживают SAX уровня 1, который довольно прост в реализации и применении. Однако с появлением новых поколений модулей становится возможной поддержка SAX2. Интерфейс SAX2 более сложен, поскольку появляются пространства имен, допускающие смешанное представление. Обработчик события элемента должен получать в свое распоряжение префикс пространства имен и локальное имя элемента. Каким образом эта информация передается посредством входных параметров? Определяется ли она в одной и той же строке, например, в виде foo:bar? Или же она распределяется в соответствии с двумя параметрами?

При обсуждении этих вопросов порождалось множество горячих дискуссий в списке рассылки *perl-xml* до тех пор, пока несколько ее участников не решили изобрести спецификацию для SAX, реализованного в стиле Perl (сейчас будет продемонстрировано, каким образом следует использовать для SAX2 новый API). Чтобы поощрить остальных членов сообщества присоединиться к этому соглашению, модуль XML: : SAX включает класс, называемый XML: : SAX: : ParserFactory. Класс Factory – это объект, единственной задачей которого является порождение объекта особого типа, в данном случае анализатора. Модуль XML: : SAX: : ParserFactory определяет удобный способ выполнения вспомогательных, рутинных операций, связанных с анализатором, например, регистрацию его опций и требований инициализации. Просто укажите требуемый вид анализатора, и в ваше распоряжение будет предоставлена соответствующая копия.

Модуль XML: : SAX реализует «уклон» в сторону совместного применения XML и Perl. Он основывается на опыте работы, включающей все лучшие возможности предыдущих модулей наряду с проведенной «работой над ошибками». С целью обеспечения реальной совместимости модулей обеспечивается базовый класс анализатора. При этом производится абстракция от большей части обычной работы, присущей всем анализаторам. Причем в обязанности разработчика входит реали-

зация уникальных свойств, характерных именно для данной задачи. Он также создает абстрактный интерфейс для пользователей анализатора, обеспечивая поддержку множества разнообразных модулей, организованных согласно реестру. Здесь выполняется индексация с применением свойств, в результате чего упрощается составление запросов на поиск необходимого модуля. Этот шаг имеет большое значение, поэтому будьте готовы к восприятию значительного объема информации и подробностей, содержащихся в этом разделе. По мнению автора, это будет полезным во всех отношениях.

Интерфейс XML::SAX::ParserFactory

Начнем с интерфейса, обеспечивающего выбор анализатора, XML: :5AX: :ParserFactory. Для тех, кто успел поработать с DBI, этот класс весьма знаком. В данном случае мы имеем дело с внешним интерфейсом для всех анализаторов, представленных в вашей системе. Достаточно запросить новый анализатор у фабрики, после чего он будет предоставлен в распоряжение пользователя. Предположим, что требуется какой-либо SAX-анализатор, применяемый совместно с пакетом обработчиков XML::SAX::MyHandler.

Рассмотрим, как выбрать анализатор, а затем воспользоваться им для чтения файла:

```
use XML::SAX::ParserFactory;
use XML::SAX::MyHandler;
my $handler = new XML: :SAX: :MyHandler;
my Sparser = XML::SAX::ParserFactory->parser( Handler => $handler );
$parser->parse_uri( "foo.xml" );
```

Получаемый в данном случае анализатор зависит от порядка, в котором установлены модули. Последний модуль (со всеми доступными возможностями, указанными с помощью параметра Requi redFeatures) будет возвращен по умолчанию. Однако возможно, что он не требуется в этой ситуации. Модуль XML: : SAX поддерживает реестр SAX-анализаторов. Каждый раз, когда устанавливается новый анализатор, производится его автоматическая регистрация в реестре, поэтому возможно получения доступа к нему с помощью ParserFactory. Если известно, что анализатор XML::SAX: :BobsParser установлен, можно потребовать его экземпляр, присвоив значение переменной $XML: :SAX: : ParserPackage следующим образом:

```
use XML::SAX::ParserFactory;
use XML::SAX::MyHandler;
my Shandler = new XML::SAX::MyHandler;
$XML::SAX::ParserPackage = "XML::SAX::BobsParser( 1.24 )";
my Sparser = XML::SAX::ParserFactory->parser( Handler => Shandler );
```

В результате присваивания переменной $XML: : SAX: ParserPackage значения XML: :SAX: :BobsParser(1.24) возвращается экземпляр пакета. При выполнении анализатора ParserFactory запрашивается метод requi re(), а затем вызывается соответствующий метод класса, new(). Значение переменной, равное 1.24, устанавливает минимальный номер версии анализатора. Если эта версия не установлена в системе, генерируется исключительная ситуация.

Чтобы получить список доступных анализаторов, предназначенных для модуля XML: : SAX, вызовите метод parsers ():

```
use XML::SAX;
my @parsers = @{XML::SAX->parsers( )};
foreach my $p ( @parsers ) {
    print "\n", $p->{ Name }, "\n";
    foreach my $f ( sort keys %{$p->{ Features }} ) {
        print "$f => ", $p->{ Features }->{ $f }, "\n";
    }
}
```

Этот метод возвращает ссылку на список хэш-данных, в котором каждый элемент содержит информацию об анализаторе, включая его имя и хэш-данные свойств. При описании предыдущей программы указывалось, что модуль XML: : SAX включает два зарегистрированных анализатора, причем каждый из них поддерживает пространства имен:

```
XML::LibXML::SAX::Parser
http://xml.org/sax/features/namespaces => 1
XML::SAX::PurePerl
http://xml.org/sax/features/namespaces => 1
```

На момент издания этой книги существовало только два анализатора, включенных в XML: : SAX. Анализатор XML:: LibXML: :SAX: : Parser или SAXAPI для библиотеки *libxml2* будет широко применяться в главе 6. С целью его применения требуется установка в вашей системе *libxml2* - скомпилированной, динамически связываемой библиотеки, написанная на языке С. Это способ является достаточно быстрым, но степень его переносимости ограничена, особенно, если не доступен соответствующий двоичный код или невозможно скомпилировать этот модуль самостоятельно. Исходя из названия следующего анализатора, XML: :SAX: : PurePerl, можно прийти к выводу, что он полностью написан на языке Perl. Здесь обеспечивается достаточная степень переносимости, поэтому возможно выполнение в любой системе, где установлен Perl. Этот начальный набор анализаторов предоставляет в распоряжение пользователя несколько различных рабочих вариантов.

Также имеет значение список свойств, соответствующий каждому анализатору, поскольку он позволяет пользователю выбирать анализатор, исходя из набора установленных критериев. Например, пусть требуется анализатор, который бы выполнял проверку достоверности и поддерживал пространства имен. Такой анализатор можно запросить посредством вызова метода require_feature() объекта factory:

```
my $factory = new XML::SAX::ParserFactory;
$factory->require_feature( 'http://xml.org/sax/features/validation' );
$factory->require_feature( 'http://xml.org/sax/features/namespaces' );
my $parser = $factory->parser( Handler => $handler );
Также можно передать конструктору factory следующую информацию:
my $factory = new XML::SAX::ParserFactory(
                Required_features => {
            'http://xml.org/sax/features/validation' => 1
            'http://xml.org/sax/features/namespaces' => 1
                }
);
my $parser = $factory->parser( Handler => $handler );
```

Если различные анализаторы успешно проходят тест, будет использован тот из них, который установлен последним. Однако, если фабрика не может найти анализатор, удовлетворяющий выдвинутым требованиям, просто генерируется исключительная ситуация.

Чтобы добавить SAX-модули в реестр, достаточно их загрузить и установить. Соответствующие установочные пакеты должны «знать» о XML: : SAX и автоматически регистрировать модули с его участием. Чтобы добавить ваш собственный модуль, можно использовать метод add_parser() модуля XML: :SAX, указав список имен модулей. Убедитесь в том, что он удовлетворяет соглашениям модулей по подклассам XML: :SAX: :Base. Затем будет продемонстрировано, каким образом можно разработать собственный анализатор и добавить его в реестр.

Интерфейс обработчика SAX2

После завершения выбора анализатора следует запрограммировать пакет обработчиков, реализующих захват потока событий анализатора, относящихся преимущественно к SAX-модулям. Модуль XML: : SAX специфицирует события и их свойства во всех подробностях и в целом. В результате обеспечивается контроль над обработчиком, благодаря чему обеспечивается полное соответствие API.

Типы поддерживаемых обработчиков событий делятся, образуя несколько групп. Знакомые нам типы включают обработчики содержимого, в том числе предназначенные для элементов и общей информации документа, распознаватели объекта и лексические обработчики, которые оперируют секциями CDATA и комментариями. DTD-обработчики и обработчики объявлений следят за всем находящимся вне элемента документа, включая элементы и объявление объектов. Модуль XML: : SAX добавляет новую группу — обработчики ошибок, перехватывающие и обрабатывающие любые исключительные ситуации, которые могут иметь место в процессе разбора.

Новый важный аспект в этом классе анализаторов состоит в том, что они распознают пространства имен, Этот процесс — одно из новшеств, привносимое SAX2. Ранее анализаторы обрабатывали уточненные имена как одно целое: комбинация префикса пространства имен и локального имени. Теперь можно разграничить пространства имен, проследить, где начинается и заканчивается его область действия, благодаря чему возникают дополнительные возможности.

Обработчики событий содержимого

А теперь сосредоточим внимание на содержимом документа, наиболее вероятно, что именно эти обработчики будут реализованы в программе обработки SAX. Отметим полезное дополнение в виде ссылки указателя документа, которое обеспечивает обработчику специальное «окно» в действиях анализатора. К новым свойствам в этом случае можно отнести поддержку пространства имен. Ниже перечислены соответствующие обработчики событий:

- **set_document_locator** (*locator*): вызывается в начале процесса анализа, анализатор использует этот метод для указания обработчику источника поступления событии. Параметр *locator* —ссылка на хэш-данные со следующими свойствами:
  - **PublicID**: общедоступный идентификатор текущего объекта, обрабатываемого анализатором;
  - **SystemID**: системный идентификатор текущего объекта, обрабатываемого анализатором;

  - LineNumber: номер строки текущего объекта, обрабатываемого анализатором;
  - ColumnNumber: последняя позиция в строке, обрабатываемой анализатором в настоящий момент времени;
- хэш-данные непрерывно обновляются последней информацией. Если обработчику не подходят передаваемые данные, и принимается решение о завершении работы, может быть выполнена проверка указателя с генерированием для пользователя значимого сообщения о том, где именно в документе-источнике была обнаружена ошибка. Для передачи указателя не требуется SAX-анализатор, хотя его присутствие было бы желательным. Следует убедиться в наличии указателя до того, как вы попытаетесь получить к нему доступ. Не пытайтесь использовать указатель где-либо, кроме как внутри обработчика событий, иначе вы получите непредсказуемый результат;
- start_document(*document*): эта программа-обработчик вызывается непосредственно после метода set_document_locator(), как только начался разбор документа. Параметр *document* — это пустая ссылка, поскольку этому событию не присущи свойства;
- end_document(*document*): последний вызываемый метод обработчика. Если обработчик достиг конца потока входных данных или встретил ошибку и завершил работу, он посылает сообщение об этом событии. Возвращенное значение для этого метода используется аналогично значению, которое возвращается с применением метода parse() обработчика. Снова параметр *document* пуст;
- start_element(*element*): везде, где анализатор встречает новый начальный тег элемента, он вызывает этот метод. Параметр *element* — это хэш-данные, содержащие свойства элемента, включая:
  - Name: строка, содержащая имя элемента, включая его префикс пространства имен;
  - Attributes: хэш-данные атрибутов, в которых каждый ключ кодируется *{NamespaceURI}LocalName*. Значение каждого элемента в хэше — это хэш-данные свойств атрибута;
  - NamespaceURI: пространство имен элемента;
  - Prefix: префиксная часть уточненного имени;
  - LocalName: локальная часть уточненного имени;

Свойства атрибутов включают:

  - Name: уточненное имя (префикс +локальная часть);
  - Value: значение атрибута, нормализованное (удалены пробелы в начале и в конце);
  - NamespaceURI: источник пространства имен.
  - Prefix: префиксная часть уточненного имени.
  - LocalName: локальная часть уточненного имени.

Свойства NamespaceURI, LocalName и Prefix задаются только в случае, если анализатор поддерживает пространства имен.

- end_element(*element*): в самом конце обрабатывается содержимое, и конечный тег элемента передается в точку, из которой анализатор вызвал этот метод. Он вызывается и для пустых элементов. Параметр *element* - это хэш-данные, содержащие следующие свойства:
  - Name: строка, содержащая имя элемента, включая его префикс пространства имен;
  - NamespaceURI: пространство имен элемента;
  - Prefix: префиксная часть уточненного имени;
  - LocalName: локальная часть уточненного имени;

Свойства NamespaceURI, LocalName и Prefix задаются только в случае, если анализатор поддерживает пространства имен;

- characters(*characters*): анализатор вызывает этот метод всякий раз, когда находит участок простого текста с символьными данными. Он может разбить этот участок на фрагменты и передавать каждый из них по отдельности, однако эти фрагменты всегда должны передаваться в том же порядке, в котором они считывались. При рассмотрении каждой отдельной части весь текст должен поступать из одного и того же объекта-источника. Параметр *characters*— это хэш-данные, содержащие одно свойство— Data, которое является строкой, содержащей символы из документа;
- ignorable_whitespace(*characters*): используется для того, чтобы описать символы пробела, появляющиеся в тех местах, где описание модели содержимого объекта специально не предусматривает символьных данных. Другими словами, символы новой строки, которые часто используются, чтобы сделать XML-код удобочитаемым посредством расположения элементов на некотором расстоянии один от другого, могут игнорироваться, поскольку на самом деле не определяют содержимое документа. Анализатор может сообщить, игнорируется ли пробел, только получив сведения из объявления DTD, и он будет его игнорировать, только если поддерживает проверку достоверности. Параметр *characters* — это хэш-данные, содержащие одно свойство, Data, которое включает символы пробела из документа;
- start_prefix_mapping(*mapping*): этот метод вызывается в случае, если анализатор обнаруживает пространство имен, входящее в область определения. Анализаторы, которые не поддерживают пространства имен, пропускают это событие, но элемент и имена атрибутов, тем не менее, содержат префиксы пространства имен. Это событие всегда происходит до начала элемента, который входит в область действия. Параметр *mapping* — это хэш-данные со следующими свойствами:
  - Prefix: префикс пространства имен;
  - NamespaceURI: ссылка URI, в которая преобразуется префикс;
- end_prefix_mapping(*mapping*): этот метод вызывается, когда пространство имен закрывается. Здесь параметр *mapping* — это хэш-данные со следующим свойством:
  - Prefix: префикс пространства имен.

Нужно обеспечить появление этого события после события конечного элемента для элемента, в котором объявлена область действия;

- processing_instruction(*pi*): эта программа обрабатывает события команд обработки из анализатора, включая те, которые найдены вне элемента документа. Параметр *pi* — это хэш-данные со следующими свойствами:
  - Target: назначение команд обработки;
  - Data: данные команды (или undef, если данных нет);
- skipped_entity (*entity*): анализаторы, не проверяющие достоверность, могут пропускать сущности, вместо того чтобы анализировать их. Например, если они не обнаружили объявление, они могут просто проигнорировать сущность вместо того, чтобы отобразить сообщение об ошибке. Этот метод предоставляет обработчику возможность произвести какие-либо действия по отношению к сущности и, возможно, даже реализовать свою схему разрешения. Если анализатор пропускает сущности, в его распоряжении окажутся следующие возможности:
  - обработать внешние параметризованные сущности (см. ссылку http://xml.org/sax/features/external-parameter-entities);
  - Обработать внешние общие сущности (см. ссылку http://xml.org/sax/features/external-general-entities);

(В XML свойства представлены в виде URI-ссылок, которые обычно могут существовать или нет. Смотрите главу 10 для получения дополнительных объяснений по поводу программы-фильтра.) Параметр *entity* — это хэш-данные со следующим свойством:

- Name: имя пропущенной сущности. Если это параметр сущности, то к имени будет присоединяться символ процента (%).

Распознаватель сущностей

По умолчанию XML-анализаторы разрешают внешние объектные ссылки, причем ваша программа может даже не знать об их существовании. Иногда необходимо переопределить это поведение. Например, у вас может быть особый способ разрешения общедоступных идентификаторов или объектов, которые являются объектами в базе данных. Какова бы ни была причина, если вы реализуете этот обработчик, анализатор запустит его до того, как попытается разрешить сущность самостоятельно.

Аргументе resolve_entity() — это хэш-данные с двумя свойствами: PublicID — общедоступный идентификатор объекта и SystemID — специфический для системы адрес, предполагающий идентичность, например, путь в файловой системе или URI-ссылку. Если общедоступный идентификатор определен как undef, то передача не выполняется, но системный идентификатор передается всегда.

Лексический обработчик событий

Реализация этой группы событий не обязательна. Возможно, вам не понадобится рассматривать эти события, поэтому не все анализаторы реализуют их передачу. Однако несколько таких событий будут иметь место. Если потребуется дублиро-

вать исходный XML-источник вплоть до комментариев и секций CDATA, то потребуется анализатор, который поддерживает эти обработчики событий.

Они включают:

- start_dtd() и end_dtd() — для разметки границ определения типа документа;
- start_entity () и end_entity() — для очерчивания области анализируемой ссылки объекта;
- start_cdata() и end_cdata() — для описания области определения секции CDATA;
- comment () — лексические комментарии, которые в противном случае будут игнорироваться анализатором.

Обработчики событий ошибок и перехват исключительных ситуаций

Модуль XML: : SAX позволяет настраивать обработку ошибок для этой группы обработчиков. Каждый обработчик принимает один аргумент, который называется исключением (исключительной ситуацией) и предназначен для подробного описания ошибок. Отдельно вызванный анализатор представляет серьезность ошибки в соответствии с W3C-рекомендациями. Существует три типа обработчиков:

- warning() : это наиболее простые обработчики исключений. Их применение означает, ошибка не столь серьезна, чтобы остановить процесс разбора. Например, ID-ссылка без согласования ID (идентификатор) вызовет отображение предупреждения, но позволит анализатору продолжить процесс разбора. Если этот обработчик не реализован, анализатор будет игнорировать исключительную ситуацию и продолжать работу;
- error() : подобный вид ошибки считается серьезным, но исправимым. К этой категории относится ошибка, связанная с нарушением достоверности. Анализатор продолжит разбор, генерируя события, за исключением того, что приложение реализует решение о выходе из системы. При отсутствии соответствующего обработчика, как правило, анализатор продолжает процесс разбора;
- fatal_error(): неисправимая ошибка может вызвать аварийное завершение работы анализатора. Анализатор не обязан продолжать работу и может лишь вывести дополнительные сообщения об ошибке. Исключением может быть синтаксическая ошибка, которая делает документ не пригодным для преобразования в XML-код, или в этом качестве может выступать объект, который нельзя разобрать. В любом случае этот пример демонстрирует наивысший уровень оповещения об ошибке, обеспечиваемый в модуле XML: : SAX.

В соответствии с XML-спецификацией предполагается, что совместимые анализаторы прекращают работу при наличии ошибок любого рода, связанных с некорректностью преобразования или недостоверностью. В Perl SAX в этом случае вызывается метод die(). Однако это еще не все. Даже после прекращения сеанса работы анализатора можно возобновить ее снова. При этом используется метод eval{}:

```
eval{ $parser->parse( $url ) };
if( $@ ) {
    # Обработка ошибок...
}
```

Свойства $@ vari able — те самые хэш-данные свойств, которые объединяют всю информацию о причинах сбоя процесса разбора.

Эти свойства включают следующее:

- Message: текстовое описание происшедшего события;
- ColumnNumber: номер символа в строке, в которой произошла ошибка в процессе синтаксического разбора;
- LineNumber: номер строки, в которой встретилась ошибка, если во время синтаксического разбора было сгенерировано исключение;
- Publ lcID: общедоступный идентификатор объекта, в котором произошла ошибка, связанная с процессом синтаксического разбора;
- SystemID: системный идентификатор, указывающий на объект, нарушающий нормальную работу в случае происшедшей в процессе разбора ошибки.

Не все исключения указывают на причину отказа в работе. Иногда анализатор вызывает исключение из-за неправильной настройки свойств.

Интерфейс XML::SAXSAX2

После того как был разработан пакет обработчиков, необходимо создать экземпляр анализатора, установить его свойства и запустить его на основе данных из XML-источника. В этом разделе обсуждается унифицированный интерфейс для анализаторов XML: : SAX

Метод parse(), которому передаются сведения о процессе разбора, получает хэш-данные опций в качестве аргументов. Можно назначить обработчики, установить свойства и определить источник анализируемых данных. Например, следующие наборы строк устанавливает как пакет обработчиков, так и документ источника для разбора:

```
$parser->parse( Handler => $handler,
                Source => { SystemId => "data.xml" } ) ;
```

Свойство Handler устанавливает общий набор обработчиков, который будет использоваться по умолчанию. Однако каждый класс обработчиков имеет свою область применения, которая будет проверяться до вызова метода Handler. Эти параметры установки включают: ContentHandler, DTDHandler, EntityResolver и ErrorHandler. Все эти параметры установки не обязательны. Если обработчик не назначен, анализатор будет просто игнорировать события и обрабатывать ошибки по-своему.

Параметр Source — это хэш-данные, которые используются анализатором для хранения всей информации о входном XML-документе. Он имеет следующие свойства:

- characterStream: этот вид дескриптора файла применяется в Perl версии 5.7.2 и выше, использующих PerlIO. При этом не выполняются преобразования кодировки. Используйте функцию read () для вычисления количества символов или функцию sysread() для определения количества байтов;
- byteStream: это свойство определяет поток считываемых байтов. Если установлено CharacterStream, описываемое свойство не применяется. Однако оно заменяет Systemld. Наряду с этим свойством должно быть установлено Encoding;

- publ icId: Это свойство не обязательно для применения, но если приложение использует общедоступный идентификатор, он хранится в Publ ic Id;
- systemId: в этой строке определяется специфическое для системы местоположение документа, например, URI или путь в файловой системе. Даже если источник является потоком данных или потоком байтов, этот параметр все равно полезен, так как может использоваться как указатель смещение ссылок внешних объектов;
- encoding: как известно, здесь хранится кодировка символов.

Любые другие опции, которые необходимо установить, находятся в наборе свойств, определенном для SAX2. Например, можно указать анализатору, что необходимо произвести какую-либо особую обработку пространства имен. Один из способов установки свойств состоит в определении свойства Features в хэш-данных опций, установленных для метода parse(). Другой способ состоит в использовании метода set_feature(). Например, рассмотрим, каким образом можно включить в проверяющий анализатор контроль достоверности, используя оба метода:

```
$parser->parse( Features => { 'http://xml.org/sax/properties/validate' => 1 } );
$parser->set_feature( 'http://xml.org/sax/properties/validate', 1 );
```

Полный перечень свойств, определенных для SAX2, смотрите в документации на web-узле **http://sax.sourceforge.net/apidoc/org/xml/sax/package-summary.html.** Можно также определить с свойства, если у вашего анализатора есть специальные атрибуты, в противном случае это сделать нельзя. Чтобы узнать, какие свойства поддерживает ваш анализатор, имейте в виду, что метод get_features() возвращает соответствующий список, а get_feature() с указанным параметром имени сообщает параметры настройки специального свойства.

Пример с драйвером

При создании простого SAX-анализатора, большая часть работы производится базовым классом XML::SAX::Base. Все что необходимо сделать, — создать подкласс этого объекта и переопределить все, что он не производит по умолчанию. Это не только удобно, но и также приводит к созданию более надежной и безопасной программы, чем если бы ее пытались написать с самого начала. Например, проверка факта реализации необходимого обработчика в данном пакете обработчиков, выполняется автоматически.

Следующий пример показывает, насколько легко создать анализатор, который работает с модулем XML: : SAX. Это драйвер, похожий на тот, который рассматривался в разделе «Драйверы для не XML-источников», кроме того, что вместо преобразования Excel-документа в XML, он читает из системного журнала web-сервера. Анализатор транслирует из системного журнала строку, похожую на эту:

```
10.16.251.137 - - [26/Mar/2000:20:30:52 -0800] "GET /index.html HTTP/1.0"
200 16171
```

в такой фрагмент XML-кода:

```
<entry>
<ip>10.16.251.137<ip>
<date>26/Mar/2000:20:30:52 -0800<date>
<req>GET /apache-modlist.html HTTP/1.0<req>
<stat>200<stat>
```

```
<size>16171<size>
<entry>
```

В листинге 5.8 представлен драйвер для web-журналов. Первая подпрограмма в пакете — это метод `parse()`. Обычно не нужно создавать собственный метод `parse()`, поскольку это делает базовый класс, но предположим, что требуется ввести некоторый XML-код, который не включает код драйвера. Поэтому заменим эту программу своей, специально созданной для обработки системных журналов web-сервера.

Листинг 5.8. SAX-драйвер web-журнала

```
package LogDriver;
require 5.005_62;
use strict;
use XML::SAX::Base;
our @ISA = ('XML::SAX::Base');
our $VERSION = '0.01';
sub parse {
    my $self = shift;
    my $file = shift;
    if( open( F, $fUe )) {
        $self->SUPER::start_element({ Name =>
        'server-log' });
        while( <F> ) {
            $self->_process_line( $_ );
        }
        close F;
        $self->SUPER::end_element({ Name =>
        'server-log' });
    }
}
sub _process_line {
    my $self = shift;
    my $line = shift;
    if( $line =~
/(\S+)\s\S+\s\S+\s\[([^\]]+)\]\s\"([^\"]+)\"\s(\d+)\s(\d+)/ ) {
    my( Sip, $date, $req, $stat, Ssize ) = ( $1, $2, $3, $4, $5 );
        $self->SUPER::start_element({ Name => 'entry' });
        $self->SUPER::start_element({ Name => 'ip' });
        $self->SUPER::characters({ Data => Sip });
        $self->SUPER::end_element({ Name => 'ip' });
        $self->SUPER::start_element({ Name => 'date' });
        $self->SUPER::characters({ Data => Sdate });
        $self->SUPER::end_element({ Name => 'date' });
        $self->SUPER::start_element({ Name => 'req' });
        $self->SUPER::characters({ Data => $req });
        $self->SUPER::end_element({ Name => 'req' });
        $self->SUPER::start_element({ Name => 'stat' });
        $self->5UPER::characters({ Data => $stat });
        $self->SUPER::end_element({ Name => 'stat' });
        $self->SUPER::start_element({ Name => 'size' });
        $self->SUPER::characters({ Data => Ssize });
        $self->SUPER::end_element({ Name => 'size' });
        $self->SUPER::end_element({ Name => 'entry' });
    }
}
1;
```

Поскольку web-журналы построчно ориентированы (один элемент в каждой строке), имеет смысл написать подпрограмму, обрабатывающую отдельные строки, —

process_line(). После этого останется только разбить элемент web-журнала на составляющие элементы и оформить их в виде XML-элементов. Метод parse () просто разобьет документ на отдельные строки и передаст их одну за другой программе-обработчику строки.

Отметим, что в пакете обработчиков обработчики событий непосредственно не вызываются. Точнее, данные передаются в базовый класс с применением программ, поскольку не нужно проверять, перехватывается ли этот тип событий пакетом обработчиков. С другой стороны, их поиск осуществляется в базовом классе, вследствие чего облегчаются задачи пользователей.

Теперь проверим анализатор в работе. Предположим, что этот модуль уже установлен (процесс установки анализатора XML: : SAX рассматривается в следующем разделе книги), поэтому достаточно просто написать программу, которая его использует. Код из листинга 5.9 показывает, каким образом можно создать пакет обработчиков и использовать его в имеющемся анализаторе.

Листинг 5.9. Программа, проверяющая работоспособность SAX-драйвера

```
use XML::SAX::ParserFactory;
use LogDriver;
my $handler = new MyHandler;
my Sparser = XML::SAX::ParserFactory->parser( Handler => Shandler );
$parser->parse( shift @ARGV );

package MyHandler;

# Инициализация объекта с опциями.
#
sub new {
     my $class = shift;
     my Sself = {@_};
     return bless( Sself, Sclass );
}
sub start_element {
     my Sself = shift;
     my $data = shift;
     print "<", $data->{Name}, ">";
     print "\n" if( $data->{Name} eq 'entry' );
     print "\n" if( $data->{Name} eq 'server-log' );
}
sub end_element {
     my Sself = shift;
     my $data = shift;
     print "<", $data->{Name}, ">\n";
}
sub characters {
    my Sself = shift;
    my Sdata = shift;
    print $data->{Data};
}
```

Модуль XML: :SAX: : ParserFactory использовался для того, чтобы показать, как может выбираться зарегистрированный анализатор. При необходимости можно определить атрибуты анализатора так, чтобы последующие запросы могли его выбирать, используя их свойства, а не имена.

Пакет обработчиков не такой уж сложный. В его задачи входит преобразование событий в поток символов XML-кода. Каждый обработчик получает ссылку

хэш-данных в качестве аргумента, посредством которого можно получить доступ ко всем свойствам объекта по соответствующему ключу. Имя элемента, например, хранится с хэш-ключем Name. Как ожидалось, это практически все, что нужно было сделать.

Установка пользовательского анализатора

Освещение возможностей модуля XML::SAX было бы не полным без демонстрации того, как создать установочный пакет, который автоматически добавляет анализатор в реестр. Добавление анализатора легко производится с помощью утилиты h2xs. Хотя первоначально эта утилита предназначалась для облегчения создания С-расширений Perl, она неоценима и в других ситуациях.

Используем ее для создания аналога установочной программы модуля, загружаемых из сети CPAN. Новый проект начнем со следующей команды:

```
h2xs -AX -n LogDriver
```

Утилита h2xs автоматически создает каталог, который называется LogDriver и включает несколько файлов:

- *LogDriver,,pm:* заглушка модуля, готового к наполнению подпрограммами;
- *Makefile.PL:* Perl-программа, которая формирует объектный файл для установки модуля;
- *test.pl:* заглушка, вместо которой может устанавливаться тестовая программа, предназначенная для проверки успешности процесса установки;
- *Changes, MANIFEST:* остальные файлы, которые помогают при инсталляции и предоставляют пользователям необходимую информацию.

Установленному модулю LogDriver.pm не требуется дополнительный код для утилиты h2xs. Нужна только переменная $VERSION, поскольку h2xs излишне «разборчив» по отношению к подобной информации.

Как известно из процесса установки модулей, полученных из сети CPAN, первое, что делается при открытии архива установочной программы, — выполняется команда perl Makefile.PM. В результате формируется файл, который называется *Makefile,* и конфигурируется программа установки для системы. Затем можно запустить объектный файл и провести установку, в результате чего модуль будет загружен в нужное место.

Любое отклонение от поведения по умолчанию установочной программы должно быть запрограммировано BMakefile.PM. Первоначально эта программа имеет следующий вид:

```
use ExtUtils::MakeMaker;
WriteMakefile(
      'NAME'          => 'LogDriver',      # Имя модуля.
      'VERSION_FROM'  => 'LogDriver.pm',   # Поиск версии.
);
```

Аргумент WriteMakeFileO — это хэш-данные свойств модуля, используемые при формировании файла Makefile. Можно включить дополнительные свойства, чтобы заставить установочную программу выполнить более сложные действия, чем обычное копирование модуля в систему. Добавим в анализатор следующую строку:

```
PREREQ_PM' => { 'XML::SAX' => 0 }
```

В результате в процессе установки инициируется проверка с целью проверки наличия в системе модуля XML::SAX. Если нет, установка прекращается и выдается сообщение об ошибке. Невозможно установить анализатор до тех пор, пока не создана среда, обеспечивающая его работу.

В Makefile.PM также необходимо добавить следующую подпрограмму:

```
sub MY::install {
     package MY;
     my $script = shift->SUPER::install(@_);
     $script =~ s/install :: (.*)$/install ::
     $1 install_sax_driver/m;
     $script .= <<"INSTALL";
     install_sax_driver :
     \t\@\$(PERL) -MXML::SAX -e
     "XML::SAX->add_parser(q(\$(NAME)))->save_parsers( )"
INSTALL
     return $script;
}
```

В этом примере в список, который поддерживает модуль XML::SAX, добавляется анализатор. Теперь можно приступить к процессу непосредственной установки модуля.

Обработка деревьев



В предыдущих главах излагались методы обработки потоков, теперь же приступим к рассмотрению другого стиля обращения с XML-данными. Вместо того чтобы разбирать фрагментированный документ с помощью XML-анализатора, можно загрузить его в память, а *затем* выполнить все необходимые действия. В результате значительно облегчается сам процесс обработки, хотя реализация подобного подхода на практике требует наличия большего количества памяти, а также приводит к увеличению загрузки центрального процессора.

В главе анализируется методика, заключающаяся в использовании устойчивых XML-объектов. Она также известна под названием *обработка деревьев (tree processing)*. В этом случае просматривается набор различных модулей и стратегий, создаются XML-деревья и определяются методы доступа к ним, включая стандартную объектную модель документа (Document Object Model, DOM). Также рассматриваются реализация быстрого доступа к внутренним разделам документа с помощью XPath и методы эффективной обработки деревьев.

XML-деревья

Каждый XML-документ можно рассматривать как набор объектов данных, образующих ациклическую структуру, которая называется деревом. Каждый объект, или *узел,* является небольшой частью документа, и представляет собой элемент, фрагмент текста или инструкцию по обработке. Один из узлов, называемый *корнем,* связан с другими узлами, которые ссылаются на следующие узлы, и т. д. вплоть до изолированных узлов. Если изобразить эту структуру в виде графа, получим картинку в виде дерева.

Использование деревьев для представления XML-документов весьма полезно. В силу ацикличной природы дерева для его обработки можно использовать последовательные методы доступа, не опасаясь «увязнуть» в бесконечном цикле. Как и в случае с деревом каталогов файловой системы, расположение любого узла можно представить в виде простой сокращенной записи. Любую ветвь дерева можно

выделить, сформировав меньшее по размеру дерево. Поэтому исходный объект подобного типа представляет собой объединение меньших по размеру деревьев, связанных в корневом узле. В результате вся информация хранится в одном месте, а поиск можно осуществлять так же, как в базе данных.

Для любого программиста возможность представления данных в виде разветвленной структуры обеспечивает упрощение решаемых задач. При обработке потоков программе-анализатору приходится запоминать множество текущих деталей, используемых для построения структур более высокого уровня или вывода информации на печать. При работе со сложным документом этот процесс может привести к путанице, а попытка объединить фрагменты информации, относящиеся к различным частям документа, — к излишнему перенапряжению. Если же работать с документом, представленным в виде дерева, вся информация может быть легко просмотрена. При этом потребуется лишь указать в тексте программы местонахождение нужного узла, и анализатор выберет требуемую информацию.

К сожалению, ничего не дается даром. За удобство доступа к любому месту документа придется расплачиваться. Во-первых, процесс построения дерева занимает много времени и сопровождается частыми обращениями к центральному процессору. Количество выполняемых операций растет в случае применения объектно-ориентированных методов. Также приходится расплачиваться дополнительным объемом памяти, поскольку при распределении объектов в дереве требуется резервировать место. При работе с большим документом (например, содержащим несколько миллионов узлов), это может привести к появлению дополнительных проблем. Тем не менее, применение деревьев оправдано (особенно в случае применения оптимизированного кода).

При описании деревьев и взаимоотношения часто упоминаются генеалогические термины. Узел, содержащий ответвления, называют *родительским (parent)* по отношению к ветвям, которые в свою очередь называют *дочерними (child)* узлами по отношению к содержащему их узлу. Соответственно термины *потомок (descendant), предок (ancestor)* и *близнец (sibling)* близки к своим традиционным значениям. Таким образом, два узла-близнеца имеют один родительский узел, а корневой узел является предком для всех узлов в дереве.

Существует несколько разновидностей деревьев, причем выбор конкретной реализации зависит от решаемой задачи. Благодаря этому реализуются различные подходы к построению модели документа. Например, объектную ссылку можно поместить в узел, отделенный от текста, или оставить ее в исходном месте. Важно уделить внимание схеме построения каждого отдельного модуля. В табл. 6.1 приведены стандартные варианты выбора разновидностей узлов.

Таблица 6.1 Определения стандартных разновидностей узлов

Тип	Свойства
Элемент	Имена, атрибуты, ссылки на дочерние узлы
Пространство имен	Префикс, URI
Символьные данные	Строка символов
Инструкции по обработке	Целевой объект, данные
Комментарии	Символьная строка
Секция CDATA	Строка символов
Объектная ссылка	Имя, текст замены (системный идентификатор идентификатор и/или общедоступный идентификатор)

В дополнение к этому набору в некоторых реализациях определяются разновидности узлов для объявлений DTD, позволяющие программисту получить доступ к определениям элементов, объектов, записей и атрибутов. Могут существовать узлы для XML-объявлений и объявлений типов документов.

Модуль XML::Simple

Простейшую модель дерева можно найти в модуле XML::Simple, разработанном Грантом Мак Лином (Grant McLean). Этот модуль предназначен для выполнения операций по обработке, чтению и сохранению файлов данных. Программисту не следует углубляться в тонкости языка XML и работы анализаторов, — достаточно просто знать, каким образом осуществляется доступ к различным структурам, используемым для хранения документов (например, к массивам или хэш-объектам). В листинге 6.1 описывается простой файл данных, который может использоваться для хранения информации.

Листинг 6.1. Файл данных программы

```
<preferences>
    <font role="default">
        <name>Times New Roman</name>
        <size>14</size>
    </font>
    <window>
        <height>352</height>
        <width>417</width>
        <locx>100</locx>
        <locy>120</locy>
    </window>
</preferences>
```

С помощью модуля XML::Simple можно осуществлять доступ к информации из файла данных. В листинге 6.2 демонстрируется, каким образом можно получить информацию об автоматически выбираемом шрифте.

Листинг 6.2. Программа, применяемая для выборки сведений о шрифтах

```
use XML::Simple;
my $simple = XML::Simple->new(); # Инициализация объекта.
my $tree = $simple->XMLin( './data.xml' ); # Считывание,
                                           # сохранение документа.
# Тестирование на предмет наличия к доступа к дереву.
print "The user prefers the font " . $tree->{ font }->{ name } . " at "
$tree->{ font }->{ size } . " points. \n";
```

Сначала инициализируется объект XML::Simple, затем вызывается встроенный метод XMLin(). В результате анализатор возвращает ссылку на корень дерева, который представляет собой иерархическую структуру, состоящую из хэш-объектов. Имена элементов являются ключами к хэш-объектам, а их значения — строками или ссылками на другие элементы хэш-объектов. Подобная организация обеспечивает удобный и быстрый доступ к любой части документа.

С целью наглядной демонстрации этой идеи рассмотрим принципы организации данных с помощью модуля Data::Dumper, позволяющего создавать последовательные структуры данных. Добавьте в конце программы следующие строки:

```
use Data::Dumper;
print Dumper( $tree );
Результат будет следующим:
$tree = {
      'font' => {
                    'size' => '14',
                    'name' => 'Times New Roman',
                    'role' => 'default'
                 },
           'window' => {
                    'locx' => '100',
                    'locy' => '120',
                    'height' => '352',
                    'width' => '417'
                 }
};
```

Переменная $tree описывает корневой элемент дерева, <preferences>. В хэш-объекте доступ к каждой записи, которая представляет собой ссылку на дочерний элемент, <font> и <window>, осуществляется через их типы. Записи передают ссылки на хэш-объекты, представляющие третий уровень элементов. В качестве подобных объектов в данном случае выступают строки, представляющие текст, который находится в реальных элементах файла. Доступ ко всему документу осуществляется с помощью простой строки, образуемой хэш-ссылками.

Приведенный пример не слишком сложен. Беспроблемное применение модуля XML::Simple объясняется тем, что здесь реализованы простейшие возможности XML. Обратите внимание на файл данных. Здесь имена элементов-близнецов ни разу не повторяются, поскольку хэш-объекты не допускают дублирования.

Если же возникает ситуация, чреватая появлением совпадающих имен, модуль XML::Simple предлагает решение. Когда элемент обладает двумя или более дочерними элементами с одинаковыми именами, используется список подобных имен в группе. Соответствующий пример можно найти в листинге 6.3.

Листинг 6.3. Более сложная программа, обрабатывающая файлы данных

```
<preferences>
     <font role="console">
          <size>9</size>
          <fname>Courier</fname>
     </font>
     <font role="default">
          <fname>Times New Roman</fname>
          <size>14</size>
     </font>
     <font role="titles">
          <size>10</size>
          <fname>Helvetica</fname>
     </font>
</preferences>
```

Здесь задача модуля XML::Simple была усложнена. В одной строке находятся три элемента <font>. Каким же образом они различаются? Обратите внимание на следующую структуру данных:

```
$tree = {
            'font' => [
```

```
                {
                    'fname' => 'Courier',
                    'size' => '9',
                    'role' => 'console'
                },
                {
                    'fname' => 'Times New Roman',
                    'size' => '14',
                    'role' => 'default'
                },
                {
                    'fname' => 'Helvetica',
                    'size' => '10',
                    'role' => 'titles'
            }
        ]
};
```

Значение переменной `font` — это ссылка на список хэш-объектов, каждый из которых моделирует один из элементов семейства <font>. Чтобы выбрать необходимый шрифт следует осуществить ряд итераций по списку до тех пор, пока не будет найден нужный элемент. Благодаря использованию подобного процесса проблема с одинаковыми именами объектов-близнецов разрешается автоматически.

В последнем файле данных некоторым элементам были назначены новые атрибуты. Здесь они являются дочерними по отношению к предшествующим элементам. Конфликт между именами атрибутов и дочерних элементов возможен, но эта проблема решается так же, как и в случае дублирования имен объектов-близнецов одного предка. Поэтому подобный подход весьма практичен.

Мы уже знаем, как направлять XML-документы в поток ввода программы, но еще не ознакомлены с операцией записи в файл. Для выполнения этой операции модуль `XML::Simple` предлагает метод, предназначенный специально для вывода XML-документов, `XML_Out()`. Можно, также, изменить существующую структуру или создать новый документ изначально, построив структуру данных, как показано выше, и затем передать ее методу `XML_Out()`.

Исходя из этих соображений, можно прийти к выводу, что модуль `XML::Simple` идеально подходит для обработки простых XML-документов; затруднения появляются при появлении более сложных элементов разметки. В частности, не могут одновременно обрабатываться дочерние элементы и текст (смешанное содержимое). Различаются только такие типы узлов, как элементы, атрибуты и текст (другими словами инструкции по обработке или секции CDATA). Так как хэш-объекты не сохраняют порядок элементов, последовательность может быть нарушена. Если ни одна из перечисленных проблем не является актуальной, можете смело воспользоваться модулем `XML::Simple`. В результате оптимизируется решение многих задач, сводятся к минимуму недостатки XML-разметки, а также обеспечивается удобный доступ к данным.

Режим дерева модуля XML::Parser

В главе 4 анализатор `XML::Parser` использовался для генерирования событий, которые управляли программами обработки потоков данных. Оказывается, этот ге-

нератор событий можно также применять для формирования деревьев! Листинг 6.4 представляет собой измененную программу считывания установок. Здесь выполняется вызов XML-анализатора XML: : Parser с целью проведения анализа и построения дерева.

Листинг 6.4. Использование анализатора XML::Parser для построения дерева

```
# Инициализация анализатора и считывание файла.
use XML::Parser;
Sparser = new XML::Parser( Style => 'Tree' );
my Stree = $parser->parsefile( shift @ARGV );

# Выгрузка структуры.
use Data::Dumper;
print Dumper( Stree );
```

При выполнении кода из листинга 6.4 на базе файла будет получен следующий результат:

```
Stree = [
          'preferences', [
               {}, 0, '\n',
               'font', [
               { 'role' => 'console' }, 0, '\n',
               'size', [ {}, 0, '9' ], 0, '\n',
               'fname', [ {}, 0, 'Courier' ], 0, '\n'
               ], 0, '\n',
               'font', [
               { 'role' => 'default' }. 0, '\n',
          'fname', [ {}, 0, 'Times New Roman' ], 0, '\n',
               'size', [ {}, 0, '14' ], 0, '\n'
               ], 0, '\n',
               'font', [
               { 'role' => 'titles' }', 0, '\n',
               'size', [ {}, 0, '10' ], 0, '\n',
               'fname', [ {}, 0, 'Helvetica' ], 0, '\n',
               ], 0, '\n',
          ]
];
```

Эта структура является более сложной, чем та, что была получена в результате применения модуля XML : : Simple. Здесь хранится много дополнительной информации, в том числе типы узлов, их порядок и смешанный текст. Каждый узел представлен одним или двумя объектами списка. Элементы структуры состоят из двух объектов: наименования и перечня его содержимого. Текстовые узлы закодированы с применением нуля и следующих за ним значений, находящихся в строке. Все атрибуты элемента хранятся в хэш-объекте в качестве первого объекта из контекстного списка. Также сохранены служебные символы, которые представляются нулем или \n. Так как списки призваны передавать содержимое элементов, порядок узлов сохраняется. Это обстоятельство имеет значение для некоторых XML-документов, например, для книг или фильмов, в которых составляющие элементы образуют некоторую последовательность.

В данном случае анализатор XML: : Parser не позволяет выводить XML-документы, как в случае с модулем XML : : Simple. Для получения завершенного универсального решения потребуется привлечь некоторые объектно-ориентированные методы.

Модуль XML::SimpleObject

Встроенные типы данных хороши во многих ситуациях, но при усложнении структуры программы для представления объектов могут понадобиться более изящные интерфейсы. Такие процедуры, как проверка типов узлов, поиск последнего дочернего элемента в дереве или замена метода представления данных без одновременного изменения оставшейся части программы, проще осуществлять с помощью объектов. Не удивляйтесь, но количество объектно-ориентированных XML-объектов не поддается точному прогнозированию.

Обзор объектных моделей для XML-деревьев начнем с модуля XML::SimpleObject, разработанного Дэном Брайаном (Dan Brian). Здесь используется структура, возвращаемая анализатором XML::Parser, выполняющимся в режиме дерева. При этом выполняется перестройка из иерархического списка в иерархию объектов. Каждый объект представляет элемент и обеспечивает методы доступа к дочерним элементам. Так же как и в случае использования модуля XML::Simple, имена элементов передаются аргументам методов, что и позволяет получить к ним доступ.

Рассмотрим, какие же преимущества связаны с применением этого модуля. В листинге 6.5 показан простой файл данных, представляющий генеалогическое дерево. Сейчас мы напишем программу, выполняющую разбор файла с построением объектного дерева, а также обход дерева с целью вывода на печать текстового описания.

Листинг 6.5. Генеалогическое дерево

```
<ancestry>
    <ancestor><name>Glook the Magnificent</name>
        <children>
        <ancestor><name>Glimshaw the Brave</name></ancestor>
            <ancestor><name>Gelbar the Strong</name></ancestor>
            <ancestor><name>Glurko the Healthy</name>
                <children>
            <ancestor><name>Glurff the Sturdy</name></ancestor>
                <ancestor><name>Glug the Strange</name>
                <children>
            <ancestor><name>Blug the Insane</name></ancestor>
        <ancestor><name>Flug the Disturbed</name></ancestor>
        </children>
        </ancestor>
        </children>
        </ancestor>
        </children>
    </ancestor>
</ancestry>
```

Текст нашей программы представлен в листинге 6.6. Сначала модуль XML::Parser разбирает файл, формируя дерево, потом передает результат конструктору модулю XML::SimpleObject. Затем процедура begat() последовательно обходит дерево и рекурсивно выводит текст. Обходя каждый элемент и предполагая, что он является предком, программа отображает его имя. Если существуют дочерние элементы (это можно узнать, проверив, возвращает ли дочерний метод определенное значение), программа опускается на нижний уровень дерева с целью выполнения соответствующей обработки.

Листинг 6.6. Программа XML::SimpleObject

```
use XML::Parser;
use XML::SimpleObject;

# Разбор файла данных и построение объекта дерева.
my $file = shift @ARGV;
my $parser = XML::Parser->new( ErrorContext => 2, Style => "Tree" );
my $tree = XML::SimpleObject->new( $parser->parsefile( $file ));

# Вывод текстового описания.
print "My ancestry starts with ";
begat( $tree->child( 'ancestry' )->child( 'ancestor' ), '' );

# Описание поколения предков.
sub begat {
        my( $anc, $indent ) = @_;

        # Вывод имени предка.
        print $indent . $anc->child( 'name' )->value;

        # При наличии потомков выполните рекурсию.
        if( $anc->child( 'children' ) and $anc->child( 'children' )-
>children ) {
            print " who begat...\n";
            my @children = $anc->child( 'children' )->children;
                foreach my $child ( @children ) {
                begat( $child, $indent . '    ' );
            }
        } else {
            print "\n";
        }
}
```

С целью проверки программы запустим ее на выполнение. Отступы в коде позволяют проиллюстрировать процесс спуска по генеалогическому дереву:

```
My ancestry starts with Glook the Magnificent who begat...
        Glimshaw the Brave
        Gelbar the Strong
        Glurko the Healthy who begat...
            Glurff the Sturdy
            Glug the Strange who begat...
                Blug the Insane
                Flug the Disturbed
```

Для доступа к данным объектов были использованы несколько методов: `child()` возвращает ссылку на объект `XML::SimpleObject`, представляющий дочерний элемент исходного узла. Метод `children()` возвращает список таких ссылок. С помощью метода `value()` осуществляется поиск узла с символьными данными и возвращает скалярную величину. При передаче аргументов этим методам ограничивается диапазон поиска теми типами узлов, которые соответствуют шаблону. Например, с помощью метода `child( 'name' )` выбираются элементы `<name>` в наборе дочерних элементов. Если поиск был неудачным, метод возвращает значение `undef`.

Описанная методика хороша для начала, однако она не годится при выполнении некоторых прикладных задач. Возможность доступа к узлам ограничена дву-

мя методами: выбором дочерних элементов или выбором их списка. Но доступ к элементу по имени не допускается, если хотя бы два элемента обладают одинаковыми именами.

К сожалению, объекты рассматриваемого модуля не располагают методами вывода на печать XML-кода, поэтому распечатка подобного документа оказывается непростым заданием. Тем не менее, начинать лучше с простого, так что изучить возможности этого модуля не помечают.

Модуль XML::TreeBuilder

Модуль XML::TreeBuilder является классом фабрики, с помощью которого строится дерево объектов XML::Element. Класс XML::Element наследует данные более старого класса XML::Element, входящего в состав пакета HTML::Tree. Поэтому можно преобразовывать файл в дерево с помощью XML::TreeBuilder, а для перемещения по нему, выборки данных из дерева или с целью изменения структуры дерева, допустимо применять методы доступа XML::Element. Обратите внимание на то, что методы доступа можно использовать для конструирования дерева по собственным правилам.

Предположим, что необходимо создать программу управления простым списком заданий (с установленными приоритетами), который хранит записи в XML-файле данных. Каждый пункт списка обладает «мгновенным» или «долгосрочным» приоритетом. Программа инициализирует список, если он пуст или если файл отсутствует. Пользователь может добавить пункты, используя переключатели -i или -l (для указания «срочного» или «долгосрочного» приоритетов). И наконец, программа обновляет файл данных и выдает результат на экран.

В первом фрагменте кода, приведенном в листинге 6.7, программа создает структуру дерева. Если файл данных найден, программа выполняет его считывание с дальнейшим построением дерева. Если же результат поиска отрицателен, дерево создается «с нуля».

Листинг 6.7. Диспетчер списка заданий, часть первая

```
use XML::TreeBuilder;
use XML::Element;
use Getopt::Std;
# Параметры командной строки.
# -i          мгновенный.
# -l          долгосрочный.
#
my %opts;
getopts( 'il', \%opts ) ;

# Инициализация дерева.
my $data = 'data.xml';
my $tree;

# Если файл существует, следует его разобрать
# и построить дерево.
if( -r $data ) {
    $tree = XML::TreeBuilder->new(  );
```

```
        $tree->parse_file($data);
    # Иначе, с самого начала создается новое дерево.
    } else {
        print "Creating new data file.\n";
        my @now = localtime;
        my $date = $now[4] . '/' . $now[3];
        $tree = XML::Element->new( 'todo-list', 'date' => $date );
        $tree->push_content( XML::Element->new( 'immediate' ) );
        $tree->push_content( XML::Element->new( 'long-term' ) );
    }
```

Добавим несколько комментариев, имеющих отношение к инициализации структуры. Минимальная структура файла данных должна быть следующей:

```
<todo-list date="DATE">
        <immediate></immediate>
        <long-term></long-temn>
</todo-list>
```

При наличии элементов <immediate> и <long-term> необходимо добавить пункты расписания. Для этого с помощью метода `new()` конструктора XML : : Element следует создать три элемента. Аргумент этого метода используется для присваивания имен элементам. Первый вызов метода содержит также аргумент `'date' => $date`, что позволяет создать атрибут с названим `"date"`. После образования узлов элементов, необходимо выполнить их связывание. Метод `push_content()` добавляет узел в список содержимого элементов.

Следующая часть программы обновляет файл данных, добавляя новый пункт, указанный пользователем. Место включения нового объекта выбирается с помощью используемой опции-переключателя (`-i` или `-l`). Для вывода XML-файла применяется метод `as_XML`, что и показано в листинге 6.8.

Листинг 6.8. Диспетчер списка заданий, часть вторая

```
# Добавление новой записи и обновление файла.
if( %opts ) {
    my $item = XML::Element->new( 'item' );
    $item->push_content( shift @ARGV );
    my $place;
    if( $opts{ 'i' }) {
        $place = $tree->find_by_tag_name( 'immediate' );
            } elsif( $opts{ 'l' }) {
        $place = $tree->find_by_tag_name( 'long-term' );
        }
    $place->push_content( $item );
}
open( F, ">$data" ) or die( "Couldn't update schedule" );
print F $tree->as_XML;
close F;
```

И наконец, эта программа выдает на терминал текущее расписание. Чтобы перейти от рассматриваемого элемента к дочернему элементу с указанным именем тега, используется метод `find_by_tag_name()`. Если под описание подходят несколько элементов, метод выстраивает их в список. Доступ к содержимому элементов в списке может осуществляться с помощью двух методов: `attr_get_i()` предназначен для атрибутов, а `as_text()` — для символьных данных. В листинге 6.9 приводится последняя часть программы.

Листинг 6.9. Диспетчер списка заданий, часть третья

```
# Вывод расписания.
print "To-do list for " . $tree->attr_get_i( 'date' ) . ":\n";
print "\nDo right away:\n";
my $immediate = $tree->find_by_tag_name( 'immediate' );
my $count = 1;
foreach my $item ( $immediate->find_by_tag_name( 'item' )) {
        print $count++ . '. ' . $item->as_text . "\n";
}
print "\nDo whenever:\n";
my $longterm = $tree->find_by_tag_name( 'long-term' );
$count = 1;
foreach my $item ( $longterm->find_by_tag_name( 'item' )) {
        print $count++ . '. ' . $item->as_text . "\n";
}
```

Чтобы проверить программу, мы включили в файл данных несколько вызовов программе (пробелы и символы табуляции улучшают восприятие кода):

```
<todo-list date="7/3">
    <immediate>
        <item>take goldfish to the vet</item>
        <item>get appendix removed</item>
    </immediate>
    <long-term>
        <item>climb K-2</item>
        <item>decipher alien messages</item>
    </long-term>
</todo-list>
```

На экране отображается следующий результат:

```
To-do list for 7/3:

Do right away:
1. take goldfish to the vet
2. get appendix removed

Do whenever:
1. climb K-2
2. decipher alien messages
```

Модуль XML::Grove

Последняя объектная модель, которая будет рассмотрена перед переходом к стандартным решениям, — модуль XML::Grove, предложенный Кеном Мак-Леодом (Ken MacLeod). Подобно модулю XML::SimpleObject, в данном случае вывод модуля XML::Parser в режиме дерева заменяется иерархической структурой объектов. Различие в том, что каждый узел представлен отдельным классом. Таким образом, для любого элемента может устанавливаться соответствие с объектом XML::Grove::Element, для инструкций по обработке — соответствие с объектом XML::Grove::PI и т. д. Текстовые элементы применяются в качестве скалярных значений.

Другая особенность этого модуля состоит в том, что объявления во внутреннем подчиненном наборе объединены в списки, доступ к которым осуществляется

с помощью объекта XML: : Grove. Каждая сущность или запись объявления доступны для подробного разбора. Например, следующая программа подсчитывает распределение элементов и других узлов, а затем выводит на экран список типов узлов и частоту их появления.

Сначала инициализируется анализатор в стиле «grove» (при этом модулю XML: :Parser сообщается о том, что для выдачи результата следует использовать модуль XML: : Parser: : Grove):

```
use XML::Parser;
use XML::Parser::Grove;
use XML::Grove;

my Sparser = XML::Parser->new( Style => 'grove', NoExpand => '1' );
my $grove = $parser->parsefile( shift @ARGV );
```

Чтобы получить доступ к содержимому объекта grove вызывается метод contents(). Он возвращает список, содержащий корневой элемент, все комментарии или внешние Pl-индикаторы. Подпрограмма tabulate() подсчитывает узлы и рекурсивно обходит дерево в направлении «сверху-вниз». Окончательно на печать выводится следующий результат:

```
# Знаки табуляции для элементов и других узлов.
my %dist;
foreach( @{$grove->contents} ) {
     &tabulate( $_, \%dist );
}
print "\nNODES:\n\n";
foreach( sort keys %dist ) {
     print "$_: " . $dist{$_} . "\n":
}
```

Здесь приведена подпрограмма, которая учитывает каждый узел в дереве, представляющий собой отдельный класс. Для получения сведений о типе узла можно применять метод ref(). В этой модели атрибуты не отождествляются с узлами, а доступ к ним осуществляется с помощью метода attributes()(как и в случае с хэш-объектами). Благодаря методу contents() процедура может продолжать обработку дочерних элементов:

```
# Для данного узла и таблицы определяется тип узла,
# добавляется учитываемое значение и, при необходмиости,
# выполняется рекурсия.
#
sub tabulate {
     my( Snode, Stable ) = @_;

     my Stype = ref( Snode );
     if( Stype eq 'XML::Grove::Element' ) {
          $table->{ 'element' }++;
          $table->{ 'element (' . $node->name . ' ) ' }++;
          foreach( keys %{$node->attributes} ) {
               $table->{ "attribute ($_)" }++;
          }
          foreach( @{$node->contents} ) {
               &tabulate( $_, Stable );
          }
     } elsif( Stype eq 'XML::Grove::Entity' ) {
```

```
        $table->{ 'entity-ref (' . $node->name . ' ) ' }++;
    } elsif( $type eq 'XML::Grove::PI' ) {
        $table->{ 'PI (' . $node->target . ' ) ' }++;
    } elsif( $type eq 'XML::Grove::Comment' ) {
        $table->{ 'comment' }++;
    } else {
        $table->{ 'text-node' }++
    }
}
```

Ниже демонстрируется типичный результат, полученный в результате использования входного XML-файла данных:

```
NODES:
PI (a): 1
attribute (date): 1
attribute (style): 12
attribute (type): 2
element: 30
element (category): 2
element (inventory): 1
element (item): 6
element (location): 6
element (name): 12
element (note): 3
text-node: 100
```

Объектная модель документа (DOM)



В этой главе мы вернемся к рассмотрению стандартных интерфейсов API совместно с объектной моделью документа (DOM). В главе 5 уже упоминалось о преимуществах, связанных с применением этих объектов, — совместимость с другими программными компонентами и гарантированное достижение требуемых результатов. Роль модели DOM в процессе обработки деревьев аналогична функциям, выполняемых SAX в случае применения потоков событий.

DOM и Perl

Применение модели DOM рекомендуется консорциумом W3C. Разработанная в качестве независимого от языка интерфейса, выполняющего представление XML-документов в памяти, модель DOM существует в различных языках (например, Java, ECMAscript[1] или Perl). В Perl имеются несколько реализаций модели DOM, в том числе воплощенные в составе модулей `XML::DOM` и `XML::LibXML`

В то время как SAX определяет интерфейс методов обработчиков, спецификация DOM поддерживает набор классов, каждый из которых выступает в роли интерфейса для методов, применяемых для обработки XML-разметки определенного типа. Следовательно, каждый экземпляр объекта управляет определенной частью дерева, обеспечивая методы доступа, позволяющие добавлять, перемещать или изменять узлы либо данные. Обычно эти объекты создаются *фабрикой объектов,* благодаря возможностям которой программисту остается всего лишь инициализировать требуемый объект.

При работе с моделью DOM каждый фрагмент XML-кода (будь это элемент, текст, комментарий или какой-либо другой элемент) следует представлять в виде объекта Node. Абстрактный класс Node расширяется с помощью конкретных классов, которые реализуют различные типы XML-разметки, в том числе Element,

[1] Язык, соответствующий общепринятым стандартам, который появился после JavaScript.

`Attr`, `ProcessingInstruction`, `Comment`, `EntityReference`, Text, `CDATASection` и `Document`. На основе этих классов создаются XML-деревья в модели DOM.

В стандартных версиях модели предусмотрены также несколько классов, которые служат контейнерами для узлов. Это удобно при перемещении фрагментов XML-кода с места на место. Примером подобного класса может служить `NodeList`, представляющий упорядоченный список узлов, например, перечень всех дочерних элементов; `NamedNodeMap` — неупорядоченный набор узлов. Объекты из этих классов часто используются в качестве аргументов методов или возвращаются обратно этими же методами. Обратите внимание на то, что рассматриваемые объекты оказывают влияние на состояние документа. Изменение одного из них немедленно влечет за собой изменение узлов в самом документе.

В ходе присваивания имен классам и методам модель DOM определяет только внешний вид реализации процесса, оставляя внутреннюю структуру на усмотрение разработчиков. В данном случае не учитываются особенности, связанные с управлением памятью, структурами данных и алгоритмами, так как они могут зависеть как от применяемого языка программирования, так и от запросов пользователя. Заказывая копию ключа слесарю, заказчик может не знать, каким образом устроен замок, ему достаточно знать, что дубликат ключа откроет замок. Аналогично спецификация внешнего вида упрощает создание расширений для наследственных модулей, согласованных со стандартом. При этом не гарантируется создание оптимального или быстродействующего кода.

Модель DOM представляет большой набор стандартных механизмов, способ применения которых сильно зависит от имеющегося уровня их совместимости. Каждый стандарт определяется на двух уровнях (скоро планируется появление третьего уровня). Версия DOM1 была выпущена в 1998 году, а версия DOM2 появилась несколько позднее. Основное различие между уровнями состоит в том, что более поздний выпуск поддерживает пространства имен. Если их не использовать, то возможностей версии DOM1 вполне достаточно.

Справочное руководство по интерфейсным классам DOM

Так как модель DOM становится наиболее предпочтительным интерфейсом при совместном использовании языков Perl и XML, изучим ее более подробно. В следующих разделах описываются отдельные интерфейсы классов, перечислены их свойства, методы и выполняемые задачи.

СОВЕТ

> В спецификации DOM определяется стандарт кодировки `UTF-16`. Тем не менее, в большинстве вариантов языка Perl стандартным образом выбирается кодировка `UTF-8`. Поэтому обработка символов, занимающих больше 8 битов, сопряжена с дополнительными трудностями. В следующих версиях DOM эта проблема будет решена путем поддержки кодировки `UTF-16`.

Класс Document

Класс Document реализует управление документом в целом, создавая при необходимости новые объекты и поддерживая информацию высокого уровня, такую как ссылки на объявление типа документа и корневой элемент.

Свойства

- *doctype* — объявление типа документа (Document Type Declaration, DTD);
- ***documentElement*** — корневой элемент документа.

Методы

- ***createElement, createTextNode, createComment, createCDATASection, create-ProcessingInstruction, createAttribute, createEntity Reference*** — генерируют новый узловой объект;
- *createElementNS, create AttributeNS (только в DOM2)* — генерируют новый элемент или узловой объект-атрибут с конкретным спецификатором пространства имен;
- ***createDocumentFragment*** — создает объект-контейнер для поддерева документа;
- ***getElementsByTagName*** — возвращает список `Node List` элементов со всех уровней документа, имеющих заданное имя тега;
- ***getElementsByTagNameNS*** *(только в DOM2)* — возвращает список `Node List` всех элементов, обладающих заданным спецификатором пространства имен и локальным именем (здесь символу звездочки (*) может соответствовать любой элемент или пространство имен, что позволяет осуществлять поиск всех элементов в указанном местоположении);
- ***getElementById*** *(только в DOM2)* — возвращает ссылку на узел, который обладает указанным атрибутом идентификатора;
- ***importNode*** *(только в DOM2)* — создает новый узел, представляющий собой копию узла из другого документа (эта операция аналогична «копированию в буфер», выполняемому при импорте разметки.

Класс DocumentFragment

Класс DocumentFragment может включать фрагмент документа. Узлы-потомки являются вершинами XML-деревьев. Этот класс отличается от класса `Document`, который обладает, как минимум, одним дочерним элементом, корнем документа и метаданными (например, типом документа). При этом содержимое класса DocumentFragment не является полностью завершенным, хотя оно и удовлетворяет критериям завершенности XML-текста (отсутствие запрещенных символов в тексте и т. п.)

Конкретные методы и свойства отсутствуют; для получения доступа к данным используются обобщенные методы доступа к узлам.

Класс DocumentType

Этот класс содержит всю информацию, заключенную в объявлении типа документа в заголовке документа, кроме данных о внешних объявлениях DTD. Поэтому класс дает название корневому элементу и всем объявленным сущностям или записям во внутреннем подмножестве объявления DTD.

Для этого класса не определены конкретные методы, свойства являются общедоступными (но доступны только в режиме считывания).

Свойства

- *name* — имя корневого элемента;
- *entities* — объект NamedNodeMap для объявлений сущностей;
- *notation* — объект NamedNodeMap для объявлений записей;
- *internalSubset (только в D0M2)* — внутреннее подмножество объявления DTD, представленного в виде строки;
- *publicId (только в D0M2)* — внешнее подмножество общедоступных спецификаторов в объявлении DTD;
- *systemId (только в D0M2)* — внешнее подмножество системных идентификаторов в объявлении DTD.

Класс Node

Все типы узлов наследуют внутренние данные класса Node. С помощью этого класса можно получить доступ к любому свойству или методу, общему для всех узлов. В некоторых случаях свойства узлов не определены, например, свойство value не определено для узла Element. Иногда в некотором программистском контексте удобно использовать обобщенные методы, например, при разработке программ для обработки узлов различных типов. С другой стороны, если известно заранее с каким типом узлов придется работать, следует использовать методы этого конкретного класса.

Все свойства, кроме nodeValue и prefix, доступны только в режиме считывания.

Свойства

- *nodeName* — свойство, определенное для элементов, атрибутов и сущностей (в контексте элементов в роли этого свойства выступает имя тега);
- *nodeValue* — свойство, определенное для атрибутов, текстовых узлов, секций CDATA, процедурных интерфейсов (PI) и комментариев;
- *nodeType* — один из следующих типов узлов: Element, Attr, Text, CDATASection, EntityReference, Entity, ProcessingInstruction, Comment, Document, DocumentType, DocumentFragment или Notation;
- *parentNode* — ссылка на узел, родительский для данного узла;
- *childNodes* — упорядоченный список ссылок на дочерние узлы данного узла (если таковые имеются);
- *firstChild, lastChild* — ссылки на первый или последний дочерний узел (если таковые имеются);
- *previousSibling, nextSibling* — узел, предшествующий рассматриваемому узлу или следующий сразу за ним, соответственно;
- *attributes* — неупорядоченный список (NamedNodeMap) узлов, являющихся атрибутами рассматриваемого класса (если таковые имеются);
- *ownerDocument* — ссылка на объект, содержащий весь документ — полезна в случае, если необходимо сгенерировать новый документ;

- *namespaceURI (только в DOM2)* — пространство имен URI, применяется, если рассматриваемый узел обладает префиксом; в противном случае возвращается пустая ссылка;
- *prefix (только в DOM2)* — префикс пространства имен, связанный с рассматриваемым узлом.

Методы

- *insertBefore* — включает узел перед указанным дочерним элементом;
- *replaceChild* — заменяет дочерний элемент новым, указанным в аргументе, и возвращает взамен старый узел;
- *appendChild* — добавляет новый узел в конец списка дочерних объектов для рассматриваемого узла;
- *hasChildNodes* — возвращает истинное значение, если узел обладает дочерними элементами; в противном случае — ложное значение;
- *cloneNode* — возвращает дубликат рассматриваемого узла. Этот метод является альтернативным и применяется для создания узлов. Все свойства сгенерированного узла идентичны свойствам предка, кроме свойства parentNode, значение которого будет неопределенным, и свойства childNodes, обладающего пустым значением. Дублированные элементы будут обладать теми же атрибутами, что и оригинальный элемент. Если аргументу deep присвоено истинное значение, копируются все потомки рассматриваемого узла;
- *hasAttributes (только в DOM2)* — возвращает истинное значение, если рассматриваемый узел обладает указанными атрибутами;
- *isSupported (только в DOM2)* — возвращает истинное значение, если используемая версия модели поддерживает указанные свойства.

Класс NodeList

Этот класс содержит упорядоченный список узлов. Он имеет динамичную природу, вследствие чего все текущие изменения структуры узлов сразу же переносятся в документ.

Свойства

- *length* — возвращает целочисленное значение, соответствующее количеству узлов в списке.

Методы

- *item* — берет целочисленное значение *n*, возвращает ссылку на *n*-й узел в списке, причем отсчет начинается с нуля.

Класс NamedNodeMap

Этот класс включает неупорядоченный набор узлов и позволяет осуществить доступ к узлам по их именам. Предусмотрена также альтернативная возможность доступа к элементам по номеру, но элементы при этом остаются неупорядоченными.

Свойства

- *length* — возвращает целочисленное значение, соответствующее числу узлов в списке.

Методы

- *getNamedItem, setNamedItem* — выбирает или добавляет узел с помощью свойства nodeName узла, используя его как ключ;
- *removeNamedItem* — возвращает узел с указанным названием, отыскивая его в списке;
- *item* — берет целочисленную величину *n,* возвращает ссылку на*n*-й узел в списке. Следует отметить, что метод не упорядочивает элементы в списке и применяется только в списках с неповторяющейся нумерацией;
- *getNamedItemNS (только в DOM2)* — отыскивает узел, используя его имя, которое действительно в пространстве имен имя (префикс пространства имен и локальное имя);
- *removeNamedItemNS (только в DOM2)* — выбирает пункт из списка, используя его имя, которое действительно в пространстве имен;
- *setNamedItemNS (только в DOM2)* — вносит узел в список, используя его имя, которое действительно в пространстве имен.

Класс CharacterData

Этот класс является расширением класса Node и обеспечивает более удобный доступ к узлам, содержащим символьные данные, а именно Text, `CDATASection`, Comment и `ProcessingInstruction`. Такие классы, как Text, наследуют свойства и методы рассматриваемого класса.

Свойства

- *data* — собственно символьные данные;
- *length* — количество символов в структуре.

Методы

- *appendData* — добавляет строку символов в конец свойства `data`;
- *substringData* — выбирает и возвращает сегмент свойства `data`, начиная с позиции, заданной величиной смещения, заканчивая в позиции, заданной величиной смещения + аргумент (от *offset* до *offset + count);*
- *insertData* — включает строку в свойство data в позиции, заданной величиной смещения *(offset)*;
- *deleteData* — присваивает свойству `data` значение, соответствующее пустой строке;
- *replaceData* — заменяет содержимое свойства `data` новой строкой, заданной аргументом метода.

Класс Element

Класс `Element` представляет наиболее часто встречающуюся разновидность узлов. Он может содержать другие узлы и узлы-атрибуты.

Свойство

- *tagname* — имя элемента.

Методы

- *getAttribute, getAttributeNode* — возвращает значение атрибута или ссылку на узел-атрибут с заданным именем;
- *setAttribute, setAttributeNode* — добавляет новый атрибут к списку элементов или заменяет существующий атрибут с таким же именем;
- *removeAttribute, removeAttributeNode* — возвращает значение атрибута и удаляет его из списка элементов;
- *getElementsByTagName* — возвращает значение свойства NodeList потомков элементов, имена которых соответствуют имени, указанному в аргументе;
- *normalize* — объединяет смежные узлы текста. Этот метод следует применять после того, как были добавлены новые текстовые узлы, что позволит обеспечить устойчивость структуры документа и не приведет к появлению фиктивных дочерних элементов;
- *getAttributeNS (только DOM2)* — отыскивает и возвращает значение атрибута, используя его имя, которое действительно в пределах списка (префикс пространства имен плюс локальное имя);
- *getAttributeNodeNS (только в DOM2)* — выбирает узел атрибута, используя уточненное имя;
- *getElementsByTagNamesNS (только в DOM2)* — возвращает список NodeList элементов, составленный из тех потомков рассматриваемого элемента, имена которых соответствуют уточненному имени;
- *hasAttribute (только в DOM2)* — возвращает истинное значение, если элемент имеет атрибут с указанным именем;
- *hasAttributeNS (только в DOM2)* — возвращает истинное значение, если рассматриваемый элемент имеет атрибут с указанным уточненным именем;
- *removeAttributeNS (только в DOM2)* — удаляет и возвращает из списка элементов узел-атрибут, используя для выбора имя, которое ограничено в пространстве имен;
- *setAttributeNS (только в DOM2)* — добавляет новый атрибут в список элементов с указанным именем, ограниченным в пространстве имен и значением;
- *setAttributeNodeNS (только в DOM2)* — добавляет к списку элементов новый узел с именем, ограниченным в пространстве имен.

Класс Attr

Свойства

- *name* — имя атрибута.
- *specified* — если программа или документ явным образом устанавливает значение атрибута, то свойство принимает истинное значение. В противном случае, если значение атрибута указано в объявлении DTD, как стандартное, присваиваемое автоматически, и не изменяется в ином месте, свойство принимает ложное значение;
- *value* — значение атрибута, представленное как текстовый узел;

- *ownerElement* (*только в DOM2*) — является элементом, к которому относится рассматриваемый атрибут.

Класс Text

Методы

- *splitText* — разбивает текстовый узел на два смежных узла, каждый из которых содержит часть текстового содержимого исходного узла. Содержимое первого узла образует часть, с начала исходного текста до символа, позиция которого задается *смещением (offset)*. Второй узел содержит оставшуюся часть исходного содержимого. Этот метод оказывается полезным в том случае, когда внутрь текстового фрагмента необходимо включить новый элемент.

Класс CDATASection

Класс `CDATASection` подобен текстовому узлу, но его содержимое защищено от вмешательства анализатора. Он может содержать символы разметки (< , &), что запрещено в текстовых узлах. Для доступа к данным следует использовать обобщенные методы класса Node.

Класс ProcessingInstruction

Свойства

- *target* — целевое назначение узла;
- *data* — назначение данных узла.

Класс Comment

Этот класс содержит узлы комментариев. Для доступа к данным следует использовать обобщенные методы класса `Node`.

Класс EntityReference

Этот класс задает объектную ссылку, определяемую узлом `Entity`. Иногда анализатор может быть настроен так, что будет разбирать все объектные ссылки на их составляющие. Если эта опция отключена, анализатор создает соответствующий узел. При выполнении разбора не применяются специальные методы, но некоторые действия с узлами могут иметь побочные эффекты.

Класс Entity

Этот класс обеспечивает доступ к сущностям в документе, используя при этом информацию из объявления DTD.

Свойства

- *publicId* — общедоступный идентификатор ресурса (используется в том случае, если сущность является внешней для документа);

- *systemId* — системный идентификатор ресурса (используется в том случае, если сущность является внешней для документа);
- *notationName* — если сущность не поддается разбору, в этом свойстве содержится информация относительно ее записи.

Класс Notation

Класс Notation представляет объявление записи, документируемое в объявлении DTD.

Свойства

- *publicId* — общедоступный идентификатор записи;
- *systemId* — системный идентификатор записи.

Модуль XML::DOM

Чтобы изучить модель DOM, применяемую в Perl, лучше начинать со знакомства с модулем XML::DOM от Энно Дерксона (Enno Derkson). Эта модель является полностью реализованной версией модели DOM первого уровня, в которую для удобства включены несколько дополнительных свойств. Модуль XML::DOM::Parser расширяет возможности модуля XML::Parser, обеспечивая построение дерева документа, встроенного в объект XML::DOM::Document, возвращая ссылку на этот объект. Эта ссылка открывает полный доступ к дереву. Все остальные модули функционируют предсказуемым образом.

Ниже приведена программа, которая использует модель DOM для обработки XHTML-файла. Она отыскивает слова «monkeys», заключенные в теги <p>, и заменяет каждое слово динамической ссылкой monkeystuff.com. Конечно, то же самое можно осуществить с помощью обычного механизма замены выражений, тем не менее, этот пример полезен. Здесь показано, каким образом, не отходя от оригинального стиля объектной модели документа, осуществляется поиск и создание новых узлов, считывание и замена значений.

В первой части программы создается анализатор, а путем вызова метода parsefile() осуществляется передача файла для разбора:

```
use XML::DOM;
&process_file( shift @ARGV );
sub process_file {
        my $infile = shift;
        my $dom_parser = new XML::DOM::Parser; # создание
                                               # анализатора
    my $doc = $dom_parser->parsefile( $infile );# передача
                                  # файла для разбора
        &add_links( $doc );     # выполнение изменений
        print $doc->toString;   # повторный вывод дерева
    $doc->dispose;              # очистка памяти
}
```

При выполнении метода возвращается ссылка на объект XML::DOM::Document, который «открывает двери» на пути к внутреннему содержимому узла. Программа передает эту ссылку процедуре add_links(), которая производит все необхо-

димые действия. И наконец, программа выдает результат, вызывая метод toStringO и разрушая объект. На последнем этапе производится необходимая очистка памяти на случай, если циркулярные ссылки между узлами могут привести к «засорению» и нехватке памяти.

В следующей части программы производится внедрение в дерево с целью начала обработки параграфов:

```
sub add_links {
        my $doc = shift;
        # Поиск всех элементов <p>.
        my $paras = $doc->getElementsByTagName( "p" );
        for( my $i = 0; $1 < $paras->getLength; $i++ ) {
            my $para = $paras->item( $i );
        # Обработка всех текстовых узлов, которые
        #являются дочерними элементами для элементов <p>.
            my @children = $para->getChildNodes;
            foreach ту $node ( @children ) {
    &fix_text($node) if($node->getNodeType eq TEXT_NODE);
        }
    }
}
```

Выполнение процедуры add_links () начинается с вызова метода объекта документа getElementsByTagName (). Она возвращает объект-список XML: : DOM: : NodeList, содержащий перечень соответствий элементу <p> в документе (вот почему многоуровневый поиск оказывается удобным механизмом), в котором выбор узлов можно осуществить по индексам, используя метод item ().

Интересующий нас в данное время фрагмент текста скрыт в текстовом узле внутри элемента <p>, поэтому, чтобы отыскать его, необходимо перебрать все дочерние элементы, выбирая и обрабатывая только текстовые узлы. Обращение к методу getChildNodes() приведет к отображению нескольких дочерних узлов, в виде оригинального списка языка Perl (если обращаться к содержимому массива) или другого объекта-списка XML: : DOM: : NodeList. В рассматриваемом примере выбран первый вариант. Тип каждого узла проверяется с помощью метода getNodeType, а результат сравнивается с константой XML: : DOM для текстовых узлов, поддерживаемой с помощью метода TEXT_NODE (). Проверенные узлы передаются процедуре с целью их дальнейшей обработки.

Последняя часть программы обрабатывает текстовые узлы, разбивая их на границе со словом «monkeys». При этом создаются ссылки:

```
sub f1x_text {
        my $node = shift;
        my $text = $node->getNodeValue:
        1f( $text =~ /(monkeys)/i ) {
# Разбивает текст на 2 текстовых узла возле слова
# monkey.
    my( $pre, $orig, $post ) = ( $', $1. $' );
    my $tnode = $node->getOwnerDocument->createTextNode( $pre );
    $node->getParentNode->insertBefore( $tnode, $node );
        $node->setNodeValue( $post );
# Включает элемент <a> между двумя узлами.
    my $link = $node->getOwnerDocument->createElement( 'a' );
    $link->setAttribute( 'href, 'http://www.monkeystuff.com/' );
    $tnode = $node->getOwnerDocument->createTextNode( $orig );
```

```
            $link->appendChild( $tnode );
            $node->getParentNode->insertBefore( Slink, Snode );
   # Рекурсивно проходит оставшуюся часть текстовых узлов
   # на случай, если искомое слово попадется еще раз.
            fix_text( Snode );
        }
   }
```

Сначала процедура получает текстовое значение узла, вызывая метод getNodeValue(). В модели D0M существуют как дополнительные методы доступа, используемые для получения или установки значений или имен, так и групповые и специализированные методы класса Node. Вместо getNodeValue() в программе можно было бы вызвать метод getData (), предназначенный специально для класса текстовых узлов. Некоторые узлы, например, элементы, не имеют определенных значений, поэтому обобщенный метод getNodeValueO возвратит неопределенный результат.

Затем программа разделяет узел на две части. При этом перед существующим текстовым узлом внедряется его двойник. После того, как текстовые значения каждого узла установлены, первый узел будет содержать все текстовое содержимое до слова «monkeys», а другой — все, что находилось после него. Следует отметить, что в приведенном примере объект XML: :DOM: :Document использовался как фабрика для создания нового текстового узла. Использованная возможность модели DOM автоматически предусматривает большое количество административных деталей, скрытых от пользователя, упрощая и обеспечивая этим безопасность процесса создания новых узлов.

На следующем шаге программа создает элемент < a > и включает его между текстовыми узлами. Как и в случае с любой полноценной ссылкой, здесь требуется найти место, чтобы включить URL, поэтому в программе используется атрибут href. С целью определения области щелчка мышью необходимо указать текст ссылки, поэтому программа создает текстовый узел со словом «monkeys» и добавляет его в список дочерних элементов. Затем процедура рекурсивно просматривает оставшийся фрагмент текстового узла на случай, если слово «monkeys» встречается несколько раз.

Работает ли данная программа? Запустите ее на выполнение, воспользовавшись следующим файлом:

```
<html>
<head><title>Why I like Monkeys</title></head>
<body><h1>Why I like Monkeys</h1>
<h2>Monkeys are Cute</h2>
<p>Monkeys are <b>cute</b>. They are like small, hyper versions of
ourselves. They can make funny facial expressions and stick out their
tongues.</p>
</body>
</html>
```

Результат выполнения программы будет следующим:

```
<html>
<head><title> Why I like Monkeys</title></head>
<bodyxhl>Why I like Monkeys</h1>
<h2>Monkeys are Cute</h2>
<p><a href="http://www.monkeystuff.com/">Monkeys</a>
```

```
are <b>cute</b>. They are like small, hyper versions of
ourselves. They can make funny facial expressions and stick out their
tongues.</p>
</body>
</html>
```

Модуль XML: :LibXML

Модуль XML::LibXML Мэтта Сержанта (Matt Sergeant) является интерфейсом библиотеки LibXML из проекта GNOME. Он быстро завоевал репутацию популярной версии модели DOM, отличаясь быстродействием и завершенностью по сравнению с более старыми версиями модулей XML::Parser. Здесь также реализована модель DOM второго уровня, благодаря чему обеспечивается встроенная поддержка пространств имен.

До сих пор мы практически не работали с пространствами имен. Большинство программистов избегает применения этих объектов. Они усложняют структуру программы, вводя дополнительные уровни сложности разметки и кода в случае, если появляется необходимость одновременно поддерживать локальные имена и префиксы. Тем не менее, с каждой новой версией XML пространства имен приобретают все большее значение, и скоро обойтись без них просто не удастся. Например, подобный объект применяется в популярном языке преобразования, XSLT. Необходимость его применения в этом случае диктуется тем, что требуется различать теги, описывающие инструкции, и теги, описывающие данные (другими словами, отображаемые элементы и те элементы, которые управляют выводом данных).

Чуть позже вы познакомитесь с пространствами имен, использующимися в HTML. Благодаря этим объектам в документ можно включать специальную разметку, например, уравнения — в обычные HTML-страницы. Язык MathML (http://www.w3.org/Math/) выполняет именно эти функции. В листинге 7.1 демонстрируется, как к документу подключить язык MathML и воспользоваться ним для поддержки пространств имен.

Листинг 7.1. Документ со включенными пространствами имен

```
<html>
<body xmlns:eq="http://www.w3.org/1998/Math/MathML">
<h1>Billybob's Theory</h1>
<p>
It is well-known that cats cannot be herded easily. That is, they do
not tend to run in a straight line for any length of time unless they
really want to. A cat forced to run in a straight line against its
will has an increasing probability, with distance, of deviating from
the line just to spite you, given by this formula:</p>
<p>
    <!-- P = 1 - 1/(x^2) -->
    <eq:math>
  <eq:mi>P</eq:mi><eq:mo>=</eq:mo><eq:mn>1</eq:mn><eq:mo>-</eq:mo>
   <eq:mfrac>
        <eq:mn>1</eq:mn>
        <eq:msup>
             <eq:mi>x</eq:mi>
             <eq:mn>2</eq:mn>
        </eq:msup>
   </eq:mfrac>
```

```
        </eq:math>
    </p>
    </body>
    </html>
```

Теги с префиксом **eq:** являются частью пространства имен, определение которого можно найти на web-узле по адресу http://www.w3.org/1998/Math/MathML. Эти теги определены в атрибуте элемента <body>. В результате использования пространства имен браузер получает информацию о том, какие теги являются оригинальными тегами HTML, а какие — внешними. Браузер, воспринимающий язык MathML, обрабатывает квалифицированные элементы в соответствии с инструкциями, предоставленными механизмом их форматирования, и не пытается применить в этом случае обычный механизм обработки языка HTML.

Некоторые браузеры не воспринимают теги MathML и отображают непредсказуемые результаты. В этом случае одной из полезных утилит оказывается программа, которая отыскивает и удаляет те ограниченные пространством имен элементы, которые могут некорректно интерпретироваться более старыми версиями процессора HTML. В следующем примере приведена программа, использующая модель D0M2, которая разбирает документ и выбирает все элементы, имеющие префикс пространства имен.

Первым делом программа разбирает файл:

```
use XML::LibXML;
my $parser = XML::LibXML->new( );
my $doc = $parser->parse_f1le( shift @ARGV );
```

На следующем шаге программа находит элемент документа и запускает рекурсивную подпрограмму, с помощью которой выполняется поиск ограниченных пространством имен элементов. После этого программа выводит документ на печать:

```
my $mathuri = 'http://www.w3.org/1998/Math/MathML';
my $root = Sdoc->getDocumentElement;
&amp;purge_nselems( $root );
print $doc->toString;
```

Следующая процедура находит узел элемента, проверяет, есть ли у него префикс, связанный с пространством имен, и в случае наличия удаляет исследуемый элемент из списка содержимого родительского узла. В противном случае, процедура пропускает элемент и переходит к обработке его потомков:

```
sub purge_nselems {
    my $elem = shift;
    return unless( ref( Selem ) =~ /Element/ );
    if( Selem->prefix ) {
        my $parent = $elem->parentNode;
        $parent->removeChild( Selem );
    } elsif( $elem->hasChildNodes ) {
        my @children = $elem->getChildnodes;
        foreach my $child ( @children ) {
            &purge_nselems( Schild );
        }
    }
}
```

Вы, наверное, заметили, что в рассматриваемую версию объектной модели документа по сравнению с традиционным интерфейсом D0M заложены некоторые ме-

ханизмы, присущие изначально языку Perl. Обращение к методу getChildnodes в контексте массива возвратило бы Perl-список вместо громоздкого списка N odeL ist. Вызванный в скалярном контексте, этот метод возвратил бы количество дочерних элементов рассматриваемого узла, вследствие чего NodeLi Sts не используется здесь вообще.

Подобные упрощения — весьма обычны в мире Perl, поэтому не будем на них останавливаться. Следует только помнить о том, что они действительно просты по сравнению с обычным объектно-ориентированным протоколом. Конечно, сразу появляется надежда на использование всеми реализациями модели DOM в языке Perl одних и тех же соглашений. Вопрос о разработке и принятии наиболее удобных стандартов для списков рассылки perl-xml возник уже довольно давно. Сейчас обсуждается вопрос о том, как наиболее логичным образом внедрить SAX2 (модуль, поддерживающий пространства имен).

Мэтт Сержант предусмотрел в пакете XML: : LibXML множество полезных механизмов. Класс Node обладает методом findnodes(), который в качестве аргумента использует XPath-выражения. Благодаря этому обеспечивается дополнительная степень гибкости при поиске узлов. Анализатор обладает опциями, которые контролируют точность его выполнения, определение сущностей и наличие нескольких значений для символов табуляции. Можно также выбрать другие специальные обработчики, предназначенные для еще не разобранных сущностей. Исходя из описанных возможностей, можно прийти к выводу, что этот модуль превосходно удовлетворяет потребности программирования с применением модели DOM.

За пределами деревьев: XPath, XSLT и некоторые другие вопросы



В предыдущей главе были введены понятия, относящиеся к обработке XML-документов как деревьев, находящихся в памяти. Они использовались довольно примитивно, — все ограничивалось построением, обходом и модификацией отдельных частей деревьев. Это актуально в случае небольших, несложных документов и задач, но серьезная XML-обработка требует более совершенных средств. В этой главе рассмотрены методы, позволяющие упростить, ускорить и повысить эффективность обработки деревьев.

Алгоритмы обхода деревьев

Вначале рассмотрим алгоритмы обхода деревьев. В соответствии с названием, они обходят дерево, находя вершины в требуемом порядке. В результате программа упрощается, а процесс обработки сосредотачивается на уровне отдельного узла. Использовать алгоритмы обхода деревьев - это все равно, что иметь дрессированную обезьянку, обученную лазить по деревьям и приносить кокосы. В этом случае вам не придется обдирать свои руки о кору деревьев, чтобы достать ценные плоды; все, что нужно сделать, - просверлить дыру в скорлупе и засунуть туда соломинку.

Простейший вид алгоритмов обхода деревьев - *итератор* (иногда называется *обходчик)*. Он может перемещаться по дереву в прямом и обратном направлениях, устанавливая ссылки на вершины, в соответствии с предписанными ему действиями. Идея продвижения по дереву состоит в установлении порядка вершин, в котором они отображаются в представлении текста документа. Точный алгоритм итерационного процесса заключается в следующем:

- если текущая вершина не задана, начать с корневой вершины;
- если у текущей вершины есть потомки, перейти к первому из них;
- если у текущей вершины есть вершина, имеющая с ней общего родителя, перейти к ней;

• если ни одна из этих ситуаций не имеет места, вернуться к списку предков текущей вершины и попытаться найти ту, чьи вершины, имеющие с ней общего родителя, остались необработанными.

С помощью этого алгоритма итератор обойдет каждую вершинув дереве, что полезно в том случае, если требуется обработать все вершины в некоторой части документа. Этот алгоритм также можно реализовать рекурсивно, однако достоинство итеративной реализации состоит в том, что в процессе обработки можно остановиться между двумя вершинами и произвести какие-либо другие действия. В листинге 8.1 показано, как можно реализовать объект итератора для DOM-деревьев. Сюда включены методы, применяемые как для продвижения по дереву в прямом и в обратном направлениях.

Листинг 8.1. Пакет DOM-итераторов

```
package XML::DOMIterator;

sub new {
     ray $class = shift;
     my $self = {@_};
     $self->{ Node } = undef;
     return bless( $self, $class );
}

# Продвижение по дереву на одну вершину вперед.
#
sub forward {
     my $self = shift;

     # Попытка спуститься на следующий уровень.
     if( $self->is_element and
             $self->{ Node }->getFirstChild ) {
         $self->{ Node } = $self->{ Node }->getFirstChild;

     # Попытка перейти к следующей вершине, имеющей общего
     # родителя, или к вершине, имеющей общего родителя
     # с предком.
     } else {
       while( $self->{ Node }) {
         if( $self->{ Node }->getNextSTbling ) {
           $self->{ Node } = $self->{ Node }->getNextSibling;
           return $self->{ Node };
         }
       $self->{ Node } = $self->{ Node }->getParentNode;
       }
     }
}
# Продвижение по дереву на одну вершину назад.
#
sub backward {
     my $self = shift;

     # Перейти к предыдущей вершине, имеющей общего родителя,
     # и спуститься к последней вершине его дерева.
     if( $self->{ Node }->getPrevious5ibling ) {
         $self->{ Node } = $self->{ Node }->getPreviousSibling;
         while( $self->{ Node }->getLastChild ) {
```

```
                $self->{ Node } = $self->{ Node }->getLastChild;
            }

        # Перейти вверх.
        } else {
            $self->{ Node } = $self->{ Node }->getParentNode;
        }
        return $self->{ Node };
}

# Вернуть ссылку на текущую вершину.
#
sub node {
        my $self = shift;
        return $self->{ Node };
}

# Установить текущую вершину.
#
sub reset {
        my( $self, $node ) = @_;
        $self->{ Node } = $node;
}

# Проверить, является ли текущая вершина элементом.
#
sub is_element {
        my $self = shift;
        return( $self->{ Node }->getNodeType == 1 );
}
```

Код из листинга 8.2 представляет тестовую программу для пакета итераторов. Она выводит краткое описание каждой вершины в дереве XML-документа в начале в прямом, а затем в обратном порядке.

Листинг 8.2. Тестовая программа, реализующая проверку пакета итераторов

```
use XML::DOM;
# Инициализация анализатора и итератора.
my $dom_parser = new XML::DOM::Parser;
my $doc = $dom_parser->parsefile( shift @ARGV );
my $iter = new XML::DOMIterator;
$iter->reset( $doc->getDocumentElement );

# Отобразить все вершины с начала до конца документа.
print "\nFORWARDS:\n";
my Snode = $iter->node;
my $last;
while( Snode ) {
        describe( Snode );
        Slast = Snode;
        Snode = $iter->forward;
}
# Отобразить все вершины с конца до начала документа.
print "\nBACKWARDS:\n";
$iter->reset( $last );
describe( $iter->node );
while( $iter->backward ) {
        describe( $iter->node );
```

```
}
# Отобразить сведения о вершине.
#
sub describe {
    my $node = shift;
    if( ref($node) =~ /Element/ ) {
        print 'element: ', $node->getNodeName, "\n";
    } elsif( ref($node) =~ /Text/ ) {
        print "other node: \"", $node->getNodeValue, "\"\n";
    }
}
```

Многие пакеты для деревьев предоставляют возможности автоматизированного обхода деревьев. В состав модуля XML::LibXML::Node включен метод iterator(), который реализует обход вершин поддерева. При этом обработка каждой вершины осуществляется с помощью соответствующей подпрограммы. Модуль Data::Grove::Visitor выполняет аналогичную функцию.

В листинге 8.3 приведен код программы, которая использует автоматизированную функцию обхода дерева для анализа команд обработки в документе.

Листинг 8.3. Программа, реализующая анализ команд обработки

```
use XML::LibXML;
my $dom = new XML::LibXML;
my $doc = $dom->parse_file( shift @ARGV );
my $docelem = $doc->getDocumentElement;
$docelem->iterator( \&find_PI );

sub find_PI {
    my $node = shift;
    return unless( $node->nodeType == &XML_PI_NODE );
    print "Found processing instruction: ", $node->nodeName, "\n";
}
```

Алгоритмы обхода деревьев крайне усложнены и могут применяться в задачах, которые включают обработку всего документа, поскольку они автоматизируют процесс перехода от вершины к вершине. Однако не всегда требуется посещать каждую вершину. Зачастую достаточно выбрать одну из группы или получить набор вершин, которые удовлетворяют некоторому критерию, например, имеют указанное имя элемента или значение атрибута. В этих случаях необходимо применять избирательный метод, что будет продемонстрировано в следующем разделе.

Язык XPath

Предположим, что в нашем распоряжении есть целая армия обезьян. Им дается команда: «Надо принести банановый напиток из кафе-мороженого на Массачусетс-Авеню севернее Porter Square». Не будучи особенно умными обезьянами, они пойдут и принесут любой попавшийся на глаза напиток, заставляя вас попробовать их все и принять нужное решение. Переобучите их, отправив в вечернюю школу изучать основы языка. Несколько месяцев спустя повторите команду. Теперь обезьяны последуют вашим указаниям, идентифицируя необходимый предмет, и принесут вам именно то, что требуется.

Мы привели описание класса задач, для решения которых предназначается XPath. Это средство реализует один из наиболее полезных методов поддержки

XML. Оно предоставляет достаточно наглядный интерфейс поиска вершин, поэтому не нужно самостоятельно разрабатывать программу их отслеживания. Достаточно указать тип требуемых вершин и XPath отыщет их. Таким образом, с помощью XML огромное, плохо организованное множество вершин превращается в четко пронумерованную картотеку, предназначенную для хранения данных.

Рассмотрим XML-документ, приведенный в листинге 8.4.

Листинг 8.4. Файл предварительных настроек

```
<plist>
    <dict>
        <key>DefaultDirectory</key>
        <string>/usr/local/fooby</string>
        <key>RecentDocuments</key>
        <array>
            <string>/Users/bobo/docs/menu.pdf</string>
            <string>/Users/slappy/pagoda.pdf</string>
            <string>/Library/docs/Baby.pdf</string>
        </array>
        <key>BGColor</key>
        <string>sage</string>
    </dict>
</plist>
```

Этот документ - типовой файл предварительных настроек программы с серией ключей данных и значений. Ничего особенно сложного в этом нет. Чтобы получить значение ключа BGColor, необходимо найти элемент <key>, содержащий слово «BGColor», и перейти к следующему элементу - <st ri ng>. Наконец, считывается значение внутри в вершине текста. В DOM можно проделать это способом, указанным в листинге 8.5.

Листинг 8.5. Программа, реализующая установку предпочтительного цвета

```
sub get_bgcolor {
    my @keys = $doc->getElementsByTagName( 'key' );
    foreach my $key ( @keys ) {
        if( $key->getFirstChild->getData eq 'BGColor' ) {
            return $key->getNextSibling->getData;
        }
    }
    return;
}
```

Написать одну такую подпрограмму довольно легко, но представьте себе, что будет в случае, если необходимы сотни подобных запросов. К тому же, она предназначена для обработки относительно простого документа, а вообразите, насколько все может усложниться при увеличении количества уровней документа. Неплохо было бы иметь короткий способ проделывать тоже скажем, в одной строке программы. Такую синтаксическую структуру было бы удобнее воспринимать, кодировать и отлаживать. Все это можно реализовать в данном случае.

Язык Xpath применяется с целью выражения пути к вершине или набору вершин в любом месте документа. Он прост, выразителен и универсален (введен консорциумом W3C, группой разработчиков, создавших XML)[1]. Вы увидите, что он

[1] Рекомендации можно найти на web-узле по адресу http://www.w3.org/TR/xpath/.

используется в XSLT для сопоставления правил с вершинами и в Xpointer-методе для связывания XML-документов с источниками. Его также можно найти во многих Perl-модулях, как будет показано далее.

XPath-выражение называется *путем размещения* и состоит из нескольких *шагов* пути, которые его корректируют с учетом приближения к цели. Начиная с абсолютной, известной позиции (например, корень документа), шаги продвижения по дереву документа направляются к вершине или к набору вершин. С точки зрения синтаксиса все это напоминает путь в файловой системе, в котором отдельные элементы разделяются символами слэша.

Следующий путь размещения демонстрирует, каким образом в предыдущем примере находится необходимое значение цвета:

```
/plist/dict/key[text()='BGColor']/following-sibling::*[1]/text( )
```

Путь размещения обрабатывается, начиная с абсолютной позиции в документе и перемещаясь в новую вершину (или вершины) на каждом шаге. В любой момент поиска текущая вершина служит контекстом для следующего шага. Если следующему шагу соответствует множество вершин, ветви поиска и процессор поддерживают набор текущих вершин. Рассмотрим, как будет обрабатываться представленный выше путь размещения:

- начинаем с корневой вершины (на один уровень выше корневого элемента);
- переходим к элементу <plist>, который является потомком текущей вершины;
- переходим к элементу <dict>, являющемуся потомком текущей вершины;
- переходим к элементу <key>, который является потомком текущей вершины и имеет значение BGColor;
- выбираем следующий элемент, находящийся после текущей вершины;
- возвращаем все текстовые вершины, принадлежащие текущей вершине.

Из-за того что поиск вершины может принять ветвящийся характер при наличии множества подходящих вершин, иногда на некотором шаге необходимо добавлять условие проверки с целью ограничения приемлемых кандидатов. Дополнительное условие проверки было необходимо на отборочном шаге для элемента <key>, когда подходило множество вершин. В результате добавилось условие проверки, требующее, чтобы элемент имел значение BGColor. Без проверки были бы получены все вершины текста для всех вершин дерева, имеющих общего предка, непосредственно следующих за элементом <key>.

Следующий путь размещения подходит для всех элементов <key> в документе:

```
/plist/dict/key
```

Различные виды условий проверки дают булевой ответ «истина/ложь». Можно проверить позицию (когда вершина есть в списке), наличие потомков и атрибутов, произвести числовые сравнения и все виды булевых выражений, использующих операции И и ИЛИ. Иногда проверка состоит только из номеров, которые являются кратким обозначением для указания индекса в списке вершин, поэтому проверка [1] означает «остановиться на первой подходящей вершине».

В квадратных скобках можно компоновать различные виды проверок совместно с булевыми операциями. В качестве альтернативы можно соединить проверки

с наборами скобок, выступающих в роли оператора И. Каждый шаг подразумевает проверку, в результате которой сокращается дерево поиска тупиков. Если на каком-либо шаге обнаруживается нулевое количество подходящих вершин, то соответствующее этой скобке действие поиска отменяется.

В соответствии с результатами проверки булевых выражений можно сформировать путь размещения с направлениями, которые называются *осями.* Ось напоминает стрелку компаса, указывающую процессору направление для перемещения. Вместо направления, заданного по умолчанию, которое опреедляет спуск от текущей вершины к ее потомкам, можно двигаться в направлении, соответствующем предку и предшествующим элементам или вершинам дерева, имеющих общего предка. Ось записывается в виде префикса к шагу с двойным двоеточием (::). В предыдущем примере использовалась ось following-sibling для перехода от текущей вершины к ее ближайшей соседке.

Шаг не ограничивается обработкой элементов. Можно указать различные виды узлов, включая атрибуты, текст, команды обработки и комментарии, или оставить его настраиваемым с селектором для любого типа вершины. Тип вершин можно указать по-разному, некоторые из них представлены в таблице:

Символ	**Соответствующий объект**
node()	Любая вершина
text()	Вершина текста
element::foo	Элемент, который называется foo
foo	Элемент, который называется foo
attribute::foo	Атрибут, который называется foo
@foo	Атрибут, который называется foo
@*	Любой атрибут
*	Любой элемент
.	Этот элемент
..	Элемент-родитель
/	Корневая вершина
/*	Корневой элемент
//foo	Элемент foo на любом уровне

Поскольку наиболее вероятно, что на шаге пути размещения будет выбран элемент, то по умолчанию тип вершины —элемент. Но есть причина, почему следует использовать другой тип вершины. В нашем примере путь размещения использовался методом text() для возвращения только вершины текста для элемента <value>.

Большинство шагов соответствуют *относительным локаторам,* поскольку они определяют, куда переходить в отношении предыдущего локатора. Хотя пути размещения включают большинство соответствующих локаторов, они всегда начинаются с *абсолютного локатора,* описывающего определенную точку в документе. Этот локатор встречается в двух разновидностях: id(), который начинается с элемента с известным атрибутом ID, и root(), который начинается с корневой вершины документа (абстрактная вершина, являющаяся предком элемента документа). Зачастую можно увидеть сокращение "/", начинающее путь и указывающее, что использовался root().

Теперь, когда обезьян научили понимать основы языка XPath, перейдем к Perl. Модуль XML::XPath, написанный Мэттом Сержантом (Matt Sergeant), в рамках популярной версии XML::LibXML - это вполне достойная реализация языка XPath. В листинге 8.6 приведена программа, которая принимает два аргумента в командной строке: файл и путь размещения XPath. Она выводит на печать текстовое значение всех вершин, которые соответствуют пути.

Листинг 8.6. Программа, использующая XPath

```
use XML::XPath;
use XML::XPath::XMLParser;

# Создать объект для разбора файла и Xpath-запросов полей.
my $xpath = XML::XPath->new( filename => shift @ARGV );

# Использовать путь из командной строки и вернуть
# список соответствий.
my $nodeset = $xpath->find( shift @ARGV );

# Отобразить каждую вершину в списке.
foreach my $node ( $nodeset->get_nodelist ) {
    print XML::XPath::XMLParser::as_string( $node ) , "\n";
}
```

Это программа устроена достаточно просто. Теперь необходим файл данных. Его код приведен в листинге 8.7.

Листинг 8.7. Файл данных XML

```
<?xml version="1.0"?>
<!DOCTYPE inventory [
    <!ENTITY poison "<note>danger: poisonous!</note>">
    <!ENTITY endang "<note>endangered species</note>">
]>
<!-- Инвентарный перечень Rivenwood Arboretum -->
<inventory date="2001.9.4">
    <category type="tree">
        <item id="284">
            <name style="latin">Carya glabra</name>
            <name style="common">Pignut Hickory</name>
            <location>east quadrangle</location>
            &endang;
        </item>
        <item id="222">
            <name style="latin">Toxicodendron vernix</name>
            <name style="common">Poison 5umac</name>
            <location>west promenade</location>
            &poison;
        </item>
    </category>
    <category type="shrub">
        <item id="210">
            <name style="latin">Cornus racemosa</name>
            <name style="common">Gray Dogwood</name>
            <location>south lawn</location>
        </item>
        <item id="104">
```

```
            <name style="latin">Alnus rugosa</name>
            <name style="common">Speckled Alder</name>
            <location>east quadrangle</location>
            &endang;
        </item>
    </category>
</inventory>
```

В ходе первого тестирования используется путь /inventory/category/item/name:

```
> grabber.pl data.xml "/inventory/category/item/name"
<name style="latin">Carya glabra</name>
<name style="common">Pignut Hickory</name>
<name style="latin">Toxicodendron vernix</name>
<name style="common">Poison Sumac</name>
<name style="latin">Cornus racemosa</name>
<name style="common">Gray Dogwood</name>
<name style="latin">Alnus rugosa</name>
<name style="common">Speckled Alder</name>
```

Находится и выводится на печать каждый элемент <name>. Будем более конкретными при указании пути /inventory/category/item/name[@style='latin']:

```
> grabber.pl data.xml "/inventory/category/item/name[@style='latin']"
<name style="latin">Carya glabra</name>
<name style="latin">Toxicodendron vernix</name>
<name style="latin">Cornus racemosa</name>
<name style="latin">Alnus rugosa</name>
```

Теперь используем атрибут ID как начальную точку для пути //item[@id='222']/note. (Если этот атрибут определен в объявлении DTD, можно использовать путь id('222')/note. В данном случае этого не выполнялось, но подобный альтернативный метод достаточно хорош.)

```
> grabber.pl data.xml "//item[@id='222']/note"
<note>danger: poisonous!</note>
```

Как извлечь теги элементов? Чтобы это сделать, нужно использовать команду:

```
> grabber.pl data.xml "//item[@id='222']/note/text( )"
danger: poisonous!
```

Когда эта инвентаризационная информация обновлялась последний раз?

```
> grabber.pl data.xml "/inventory/@date"
    date="2001.9.4"
```

Благодаря XPath можно многое сделать! Вот путь, который проходит по дереву глупая обезьяна:

```
> grabber.pl data.xml "//*[@id='104']/parent::*/preceding-sibling::*/
child::*[2]/name[not(@style='latin')]/node( )"
Poison Sumac
```

Обезьяна начинала с элемента с атрибутом id='104', поднималась уровнем выше, перепрыгивала на предыдущий элемент, отступала на второй элемент-потомок, находила <name>, атрибуту типа которого присваивалось значение 'latin', и перепрыгивала к потомку элемента, оказывающегося вершиной текста со значением Poison Sumac.

Отсюда видно, как следует использовать Xpath-выражения, чтобы выбрать и вернуть набор вершин. Рассмотренная реализация является более мощной. Оригиналь-

ный модуль Майкла Родригеса (Michel Rodriguez), XML: :Twig, реализован в духе Perl с точки зрения применения XPath-выражений. Здесь используются хэш-данные с целью преобразования в подпрограммы. Благодаря этому могут реализовываться функции, автоматически вызываемые для определенных типов вершин.

Приведенная в листинге 8.8 программа иллюстрирует, как это все реализовано на практике. При инициализации объекта XML: :Twig можно установить группу обработчиков в хэше, где ключами будут XPath-выражения. На протяжении этапа разбора эти обработчики вызываются для соответствующих вершин, аналогично тому, как строится дерево.

Рассмотрим листинг 8.8. Отметим, что символы «коммерческое at» (@) теряются. Это происходит потому, что наличие символа @ может привести к путанице с XPath-выражениями, существующими в Perl-контексте. В XPath @foo ссылается на атрибут, который называется foo, а не на массив с названием foo. Об этих особенностях следует помнить, когда рассматриваются соответствующие примеры из этой книги и пишутся XPath-программы для Perl; следует избегать символов @, поэтому Perl не пытается видоизменять массивы в середине выражений.

Если программа выполняет столько действий с массивами Perl и ссылками атрибутов XPath, что не понятно, какой символ @ к чему относится, то рассмотрите ссылку на атрибуты без сокращений, используя ось «атрибута» XPath: attribute::foo. Это порождает проблему двойных двоеточий и их различной семантики в Perl и XPath. Тем не менее, поскольку в XPath существуют несколько жестко запрограммированных осей, и они всегда изображаются с применением строчных символов, их легче отличить с первого взгляда.

Листинг 8.8. Иллюстрация действий обработчиков

```
use XML::Twig;

# Буферы для хранения текста.
my $catbuf = '';
my $itembuf = '';

# Инициализация анализатора с обработчиками для
# оперирования с вершинами.
my $twig = new XML::Twig( TwigHandlers =>   {
                              "/inventory/category" => \&category,
                              "name[\@style='latin']" => \&latin_name,
                              "name[\@style='common']" => \&common_name,
                              "category/item" => \&item,
                                  }):

# Разбор, обрабатываются попадающиеся по пути вершины.
$twig->parsefile( shift @ARGV );

# Обработка категории элемента.
sub category {
     my( $tree, $elem ) = @_;
     print "CATEGORY: ", $elem->att( 'type' ), "\n\n", Scatbuf;
     Scatbuf = '';
}

# Обработка объекта элемента.
sub item {
     my( $tree, $elem ) = @_;
```

```
        $catbuf .= "Item: " . $elem->att( 'id' ) . "\n" . $itembuf . "\n";
        $itembuf = '';
    }

    # Обработка имени, набранного с применением латинских букв.
    sub latin_name {
        my( $tree, $elem ) = @_;
        $itembuf .= "Latin name: " . $elem->text . "\n";
    }

    # Обработка общего имени.
    sub common_name {
        my( $tree, $elem ) = @_;
        $itembuf .= "Common name: " . $elem->text . "\n";
    }
```

Эта программа получает файл данных, как показано в листинге 8.7, и выводит отчет. Отметим, что поскольку обработчик вызывается только после завершения полного построения элемента, общий порядок вызовов обработчиков может оказаться не тем, который ожидался. Обработчики для потомков вызываются до обработчиков для предков. По этой причине необходимо буферизовать вывод и классифицировать его в соответствующий момент.

Результат выглядит следующим образом:

```
CATEGORY: tree

Item: 284
Latin name: Carya glabra
Common name: Pignut Hickory

Item: 222
Latin name: Toxicodendron vernix
Common name: Poison Sumac

CATEGORY: shrub

Item: 210
Latin name: Cornus racemosa
Common name: Gray Dogwood

Item: 104
Latin name: Alnus rugosa
Common name: Speckled Alder
```

С помощью XPath чрезвычайно упрощается задача поиска и описание типов обрабатываемых вершин в документе. Существенно сокращается объем создаваемой программы, поскольку остается только обойти дерево, чтобы проверить различные части. Кроме того, все это легче прочитать, чем запрограммировать. И поскольку XPath, по сути, представляет стандарт, он может применяться в самых различных модулях.

Язык XSLT

Если XPath рассматривается с позиций обычного синтаксиса выражений, то XSLT - это механизм, реализующий подстановку шаблона. Язык XSLT - это основанный на XML язык программирования, описывающий преобразования раз-

личных типов документов. С помощью XSLT можно делать многое, например, описывать, как преобразовать любой XML-документ в HTML-код или свести в таблицу совокупность данных в таблице XML-формата. На самом деле, возможно, даже не понадобится программировать на Perl или каком-либо другом языке программирования. Все, что будет необходимо, - XSLT-сценарий и один из множества механизмов преобразования, доступных для XSLT-обработки.

ИСТОКИ XSLT

Язык XSLT основан на языке стилей XML: Transformations (преобразования). Название гласит, что в данном случае мы имеем дело с компонентом языка стилей XML (XML Style Language, XSL), предназначенного для выполнения задач по преобразованию входных данных XML в специальный формат, называемый XSL-FO (FO основан на « объектах форматирования»). Формат XSL-FO включает как содержимое, так и команды, выполняющие вывод информации на экран.

Хотя аббревиатура XSL является составной частью аббревиатуры рассматриваемого языка, XSLT— нечто большее, чем просто этап форматирования; он является важным инструментом XML-обработки, который упрощает процесс преобразования одной разновидности XML в другую или преобразование XML в текст. По этой причине комитет W3C (который также является творцом XSLT) разработал рекомендации для этого языка задолго до того, как был определен формат XSL в целом.

Ознакомиться со спецификациями и найти ссылки на учебники по XSLT можно на web-узле **http://www.w3.org/TR/xslt.**

XSLT-сценарий преобразования является XML-документом. В основном он содержит правила, которые называются *шаблонами* и указывают на то, как следует обрабатывать отдельный тип вершин. Обычно шаблоны выполняют две вещи: описывают то, что необходимо получить в результате, и определяют, как следует выполнять обработку.

Рассмотрим сценарий, представленный в листинге 8.9.

Листинг 8.9. Таблица стилей XSLT

```
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">

     <xsl:template match="html">
          <xsl:text>Title: </xsl:text>
          <xsl:value-of select="head/title"/>
          <xsl:apply-templates select="body"/>
     </xsl:template>
     <xsl:template match="body">
          <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="hl | h2 | h3 | h4">
          <xsl:text>Head: </xsl:text>
          <xsl:value-of select="."/>
     </xsl:template>

     <xsl:template match="p | blockquote | li">
          <xsl:text>Content: </xsl:text>
          <xsl:value-of select="."/>
     </xsl:template>
</xsl:stylesheet>
```

С помощью этого сценария преобразования HTML-документ перекодируется в ASCII-коды с включением некоторых особых меток в тексте. Каждый элемент `<xsl:template>` – это правило, которое соответствует части XML-документа.

Его содержимое включает команды для XSLT-процессора, описывающие, что необходимо отобразить на экране. Директивы вида <xsl:apply-templates> управляют обработкой других элементов (как правило, потомков). Мы не будем вдаваться в подробности синтаксиса языка XSLT, поскольку читателю доступны все книги по этой теме. Мы намерены показать, как можно сочетать XSLT с Perl для выполнения эффективного прослеживания XML-документов.

У читателя может возникнуть вполне логичный вопрос: «Зачем для преобразования XML понадобилось использовать другой язык, если, как известно, можно делать то же самое с помощью Perl?» Правильно, средствами XSLT можно сделать все то, что достигается с помощью Perl. Его ценность заключается в простоте. Язык XSLT можно изучить за несколько часов, а чтобы освоить Perl, понадобится гораздо больше времени. Исходя из опыта разработки программного обеспечения для XML, мы считаем удобным, использовать XSLT в качестве конфигурационного файла, который не мог бы самостоятельно поддерживать ни один программист. Таким образом, вместо того, чтобы рассматривать XSLT как конкурента Perl, считайте его дополнительной технологией, которая при необходимости может обеспечить доступ посредством Perl.

Как Perl-разработчики используют в своих программах возможности XSLT? В листинге 8.10 показано, как выполнить XSLT-преобразование в документе, используя XML::LibXSLT - интерфейс Мэтта Сержанта, ориентированный на быстродействующую библиотеку GNOME (имя библиотеки — LibXSLT) как одного из нескольких XSLT-решений, доступных среди набора инструментов в сети CPAN[1].

Листинг 8.10, Программа выполняет XSLT-преобразование

```
use XML::LibXSLT;
use XML::LibXML;

# Аргументы для этой команды - это таблица стилей и
# файлы-источники.
ту( $style_file, @source_files ) = @ARGV;

# Инициализация анализатора и XSLT-процессора.
my Sparser = XML::LibXML->new( );
my $xslt = XML::LibXSLT->new( );
my $stylesheet = $xslt->parse_stylesheet_file( $style_file );

# Для каждого файла источника выполняется разбор,
# преобразование и вывод на печать результата.
foreach my $file ( @source_files ) {
     my $source_doc = $parser->parse_file( $source_file );
     my $result = $stylesheet->transform( $source_doc );
     print $stylesheet->output_string( $result );
}
```

Преимущество этой программы состоит в том, что она анализирует таблицу стилей лишь один раз, сохраняя ее в памяти для повторного использования при работе с другими документами-источниками. Впоследствии, при необходимости, будет доступно дерево документа для дальнейшей обработки в случае:

[1] Доступны также и другие инструменты, например, Perl-модуль XML: :XSLT, а также модуль XML:: SabXotron, основанный на С-библиотеках Expat и Sablotron (последняя библиотека фактически является XSLT-библиотекой, разработанной фирмой Ginger Alliance: http://www.gingerall.com).

- выполнения заключительной и предварительной обработки текста документа с процедурами поиска-замены;
- выделения части документа для дальнейшего преобразования только этой части;
- запуска итератора для обработки некоторых вершин дерева, что затруднительно в случае с XSLT.

Существует бесчисленное количество возможностей и, как это бывает в Perl, в любом случае можно использовать несколько способов.

Оптимизированная обработка деревьев

Большой недостаток при использовании деревьев для XML-обработки проявляется в огромных затратах памяти и процессорного времени. Возможно, это не так очевидно в случае небольших документов, но становится заметным, как только количество вершин в документах вырастает до многих тысяч. Обычной книге, имеющей несколько сотен страниц, могут соответствовать десятки тысяч вершин. Каждая из них требует выполнения размещения объекта, реализуемого в виде процесса, который занимает значительное время и требует много памяти.

Хотя, возможно, для выполнения задачи не потребуется строить все дерево. Может потребоваться только незначительный сегмент дерева, внутри которого будет успешно выполнена вся необходимая обработка. В этом случае следует воспользоваться преимуществом оптимизированного метода разбора, включенного в состав модуля XML: :Twig (напомним, что этот модуль рассматривался ранее в разделе "XPath"). Этот метод позволяет заранее установить, какие части (или «ветви») дерева будут обрабатываться для того, чтобы только эти части строились. В результате получаем гибрид дерева. В этом случае при обработке событий достигается оптимизация производительности по скорости и времени.

В модуле XML: : Twi g выполняются три режима операций: традиционный режим дерева, похожий на тот, который рассматривался до настоящего времени; режим «кусок», при реализации которого строится все дерево, но в текущий момент времени хранится в памяти только фрагмент дерева (отчасти напоминает память со страничной организацией); режим со множеством корней, когда строятся несколько выбранных ветвей дерева.

В листинге 8.11 продемонстрированы возможности модуля XML: :Twig в « кусочном» режиме. Данными для этой программы является книга DocBook с несколькими элементами <chapter>. Настоящие документы могут достигать огромного размера, иногда сотен Мбайт и более. Программа разбивает обработку по главам для того, чтобы требовался только фрагмент области памяти.

Листинг 8.11. Программа разбиения дерева на куски

```
use XML::Twig;

# Инициализация ветви, разбор и отображение
# полученной ветви.
my $twig = new XML::Twig( TwigHandlers => { chapter => \&process_chapter });
$twig->parsefile( shift @ARGV );
$twig->print;

# Обработчик главы: обрабатывает главу, а затем выполняет
```

```
# очистку.
sub process_chapter {
     my( $tree, $elem ) = @_;
     &process_element( $elera );
     $tree->flush_up_to( $elem ); # Устранение этой строки
# Приводит к напрасным затратам памяти.
}

# Добавляет 'foo' к имени элемента.
sub process_element {
     my $elem = shift;
     $elem->set_gi( $elem->gi . 'foo' );
     my @children = $elem->children;
     foreach my $child ( @children ) {
          next if( $child->gi eq '#PCDATA' );
          &process_element( $child );
     }
}
```

Программа изменяет имена элементов, добавляя к ним строку "foo". Изменение имен - это лишь пустяковая работа, необходимая для того, чтобы проконтролировать использование памяти. Обратите внимание на строку в функции process_chapter():

```
$tree->flush_up_to( $elem );
```

Благодаря этой команде экономится память. Без нее было бы построено все дерево, которое хранилось бы в памяти до тех пор, пока документ не был выведен на печать. Но, когда она вызывается, дерево, построенное до данного элемента, удаляется, а его текст выводится (называется очистка). Процент использования памяти никогда не превышает величину, требуемую для хранения самой большой главы в книге.

Чтобы проверить эту теорию, запустим программу вместе с документом, объем которого составляет 3 Мбайт. Сначала заключите эту строку в знаки комментария, затем раскомментируйте ее. Без применения очистки область динамической памяти программы увеличилась до 30 Мбайт. Удивительно, как много памяти необходимо объектно-ориентированному процессору деревьев; в нашем случае в десять раз больше, чем размер файла. Но с применением очистки объем требуемой памяти программы колебался вокруг лишь нескольких Мбайт; экономилось около 90% системных ресурсов. В обоих случаях было построено все дерево, поэтому суммарное время обработки оставалось тем же. Чтобы экономить время центрального процессора так же, как и память, необходимо использовать режим со множеством корней.

Многокорневой режим реализуется до выполнения анализа корней создаваемых ветвей. Значительно экономятся время и память, если ветви по размеру меньше, чем весь документ. В примере «кусочного» режима не многое можно было бы сделать, чтобы ускорить процесс, поскольку сумма элементов <chapter> приблизительно такая же, как размер документа. Поэтому рассмотрим пример, который лучше иллюстрирует многокорневой режим.

Программа, представленная в листинге 8.12, считывает документы DocBook и выводит заголовки глав, то есть оглавление. Чтобы получить эту информацию, необходимо построить дерево для всей главы, выраженное записями XPath, «глава-заголовок».

Листинг 8.12. Разветвленная программа

```
use XML::Twig;

my $tw1g = new XML::Twig( TwigRoots => { 'chapter/title' => \&output_title });
$twig->parsefile( shift @ARGV );

sub output_title {
    my( $tree, $elem ) = @_;
    print $elem->text, "\n";
}
```

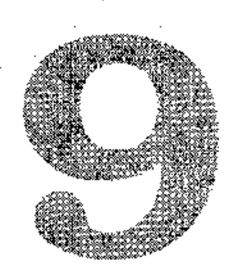
Здесь ключевая строка - это строка, которая содержит ключевое слово TwigRoots. Она принимает хэш-данные обработчиков, а процесс ее функционирования весьма напоминает процесс выполнения обработчиков TwigHandlers, рассмотренных ранее. Отличие состоит в том, что вместо построения всего дерева документа, программа строит только те деревья, чьи корни являются элементами <ti tle>. Это небольшой фрагмент целого документа, поэтому можно ожидать, что будет сэкономлено больше времени и памяти.

Насколько больше? Запустив программу на основе тех же тестовых данных, видим, что использование памяти едва достигает 2 Мбайта, а общее время обработки - 13 секунд. Сравните этот результат с 30 Мбайтами памяти (размер, обязательный для построения всего дерева) и суммарным временем, необходимым для получения заголовков. Экономия ресурсов существенная, как памяти, так и времени центрального процессора.

В результате применения модуля XML: : Twi g может достигаться значительный рост производительности программ обработки деревьев, но необходимо знать, в каких случаях помогает разбиение дерева на куски, а в каких - применение многочисленных корней. Вы не сэкономите много времени, если сумма ветвей почти такая же, как размер документа. Разбиение дерева на куски бесполезно до тех пор, пока куски существенно меньше, чем сам документ.

R S S , S O A P

и некоторые другие XML-приложения



В этой и последующей главах реализуется практическое приложение изложенной ранее теории. Здесь начинается «царство» XML-приложений, предназначенных для обработки документов, причем в роли фундамента выступают анализаторы. Не удовлетворяясь указанием элементов и атрибутов, «живущих» лишь один день, эти инструменты высокого уровня «расшифровывают» значение всей структуры. При этом используются встроенные в данные инструменты директивы.

Если упоминаются XML-приложения, имеются в виду форматы XML-документов, а не компьютерные программы (приложения другого вида), предназначенные для обработки этих документов. Обратимся к некоторым примерам. Предположим, что вам встретилось утверждение следующего вида: «XML-приложение GreenMonkeyML обеспечивает семантическую разметку для зеленых мартышек». Посетив начальную страницу этого проекта по адресу http://www.greenmonkey-markup.com, мы найдем документацию по форматам, примеры документов, рекомендации по применению, документы DTD либо схемы, применяемые для проверки документов GreenMonkeyML, а также интерактивные инструменты проверки достоверности. Все это укладывается в определение XML-приложения.

Предметом рассмотрения этой главы являются XML-приложения, которые занимают почетное место в качестве общедоступных Perl-модулей.

XML-модули

Термин *XML-модуль* на самом деле носит локальный характер и отличается от Perl-модулей в сети CPAN, обеспечивающих отсылку электронной почты, обработку изображений либо реализующих компьютерные игры. Однако и в этом случае вам предоставляется достаточно широкий выбор возможностей. До сих пор производилось исчерпывающее исследование Perl-расширений, с помощью которых выполняются общие операции по обработке XML-документов, но не рассматривались какие-либо дополнительные вопросы. В вашем распоряжении оказались фрагменты бессодержательного XML-кода, судьбу которых еще предстоит решить. Многие при-

меры, рассматриваемые в книге, фактически являются программами, выполняющими следующее: вызывается обрабатывающий документ XML-анализатор, затем соответствующим образом изменяются значения элементов и атрибутов.

Возможности рассматриваемых здесь модулей превосходят то, что может обеспечить обобщенное семейство модулей, выполняющее синтаксический разбор и обработку, которое создается на основе одного из анализаторов. Эти модули поддерживают API, с помощью которых обеспечивается привязка к исходному XML-коду с выделением методов и процедур, специфичных для реализуемого XML-приложения.

Perl-модули, обладающие признаками XML-приложений, можно разделить на три класса. В этой и последующей главах будет рассмотрен пример каждого из них, а в следующей главе вы сможете сами разработать соответствующий модуль.

Модули-помощники, как правило, наиболее простые и компактные. Зачастую они являются более компактными, чем оболочки исходных XML-процессоров, но их важность поистине неоценима. И эта ценность наглядно проявляется в том случае, когда требуется разработать несколько программ, выполняющих операции считывания и записи для XML-документа с определенным форматированием. Модуль-помощник поддерживает общие методы, благодаря чему программист может не заботиться о соблюдении формата документов приложений или формальной корректности генерируемых выходных данных.

Категория XML-помощников, облегчающих процесс программирования, описывает Perl-расширения, которые используют XML с целью реализации некоторых «изюминок» в вашей программе, даже если входные либо выходные данные мало связаны с XML. Наиболее важные примеры реализуются средствами DBI-подобного модуля `XML::SAX` с помощью семейства PerlSAX2, а также посредством отдельных инструментов. (Например, модуль `XML::Generator::DBI`, который реализован на стыке баз данных и SAX-обработки, выполняемой с помощью существующих Perl-модулей.)

И наконец, существуют программы, использующие XML, Которые обладают некоторыми особенностями. Эти особенности проявляются наличием уровней абстракции между предназначением данных программ и базовым XML-кодом. Вызов таких программ внутри XML-приложения напоминает вызов Microsoft Word в среде С-приложения. Например, при работе с модулем `SOAP::Lite` создаются документы, предназначенные для конечного пользователя. Эти документы хранятся в памяти до тех пор, пока не осуществится подключение к Интернету с применением протокола HTTP; в этом случае роль XML является совершенно очевидной.

Модуль XML::RSS

Благодаря *модулям-помощникам* возможно применение специализированных версий XML-процессоров, входящих в состав вашего набора инструментов Perl и XML. С этой точки зрения `XML::Parser` и подобные ему модули можно рассматривать в качестве приложений-помощников, поскольку их применение существенно облегчает задачи программиста. В частности, вам не придется выполнять вручную операции по обработке XML-кода, поскольку существуют встроенные Perl-функции, выполняющие считывание данных из файла, и регулярные выражения, позволяющие преобразовывать документы в пригодные к применению объекты либо пото-

ки событий. Модуль XML::Writer и родственные ему модули позволяют вместо традиционных операторов печати воспользоваться более абстрактным и безопасным путем с целью создания XML-документов.

Следует учитывать то, что применение рассматриваемых XML-модулей осуществляется с целью выполнения весьма специфических задач. При использовании одного из модулей в вашей программе нетрудно прийти к выводу, что можно спланировать применение отдельных четко определенных подсекций XML-кода, а не всего кода в целом. Учитывая это ограничение, можно применять (и создавать) программные модули, предназначенные для обработки исходного XML-кода. В этом случае основная часть кода представляется с помощью методов и процедур, специфичных для применяемых приложений.

В качестве примера рассмотрим модуль XML: :RSS — компактный счетчик, разработанный Джонатаном Айзенцопфом (Jonathan Eisenzopf).

Начальные сведения о RSS

Аббревиатура RSS (Rich Site Summary (Обзор богатых сайтов) либо Really Simple Syndication (Просто синдикат владельцев недвижимости), конкретная расшифровка зависит от вашей фантазии), обозначает одно из первых XML-приложений, быстро завоевавших популярность благодаря широкому распространению Сети. Во-первых, RSS обеспечивает согласованный способ резюмирования содержимого web-страниц, но этим его возможности не ограничиваются. Благодаря его применению администраторы серверов новостей, web-журналов и других, часто обновляемых web-узлов получают простой стандартный метод оповещения мира о происходящих событиях. Программы, выполняющие синтаксический разбор RSS, могут реализовывать любые действия по отношению к этому документу. В частности, результаты синтаксического разбора могут отсылаться разработчикам по электронной почте либо посредством web-страниц. Сумматор — специальный тип RSS-программы, предназначенный для сбора RSS-сведений из различных источников. Собранная информация включается в состав новых RSS-документов, благодаря чему облегчается выполнение программы синтаксического разбора RSS-документов.

В настоящее время распространены два сумматора: Netscape, применяемый с помощью настраиваемого узла my.netscape.com (фактически здесь находится «место рождения» первых версий RSS-программ) и сумматор Дэйва Винера (Dave Winer), находящийся на web-узле http://www.scripting.com (общедоступный интерфейс этого сумматора находится на web-узле http://aggregator.userland.com/register). Эти сумматоры могут применяться в качестве источников RSS-документов. Причем они представляют собой нечто вроде «универсального RSS-магазина», на полках которого можно найти много интересных вещичек. Сумматоры применяются web-узлами, выполняющими сбор сведений и установку ссылок на новые информационные ресурсы в Web. В результате, пользователи web-узлов, поддерживающих RSS, могут получать нужные им сведения. Примером подобного узла может служить Meerkat (http://meerkat.oreillynet.com), размещенный в сети O'Reilly.

Применение модуля XML::RSS

Модуль XML: : RSS весьма полезен для конечного пользователя. Он применяется для синтаксического разбора RSS-документов, а также позволяет разрабатывать

собственные RSS-документы. Естественно, вы можете комбинировать предоставленные вам возможности с целью синтаксического разбора документов, их изменения, а также с целью их повторной записи. Этот модуль применяет простую и хорошо документированную объектную модель с целью представления документов в памяти, как в случае с недавно рассмотренными модулями, основанными на применении деревьев. Подобный вид XML-помощников можно рассматривать в качестве усложненной версии привычного XML-инструмента общего назначения.

В последующих примерах рассматривается воображаемый web-журнал, реализованный в виде часто обновляемого журнала или персональной колонки, публикуемых в Сети. Благодаря применению RSS в составе web-журналов можно быстро составлять резюме по последним записям в составе единого RSS-документа.

Обратите внимание на следующие записи в web-журнале (случайным образом выбранные из пула воображаемых концепций). И, в первую очередь, на то, как выглядят эти записи в окне web-браузера:

```
Oct 18, 2002 19:07:06

Today I asked lab monkey 45-X how he felt about his recent chess
victory against Dr. Baker. He responded by biting my kneecap. (The
monkey did, I mean.) I
think this could lead to a communications breakthrough. As well as
painful swelling, which is unfortunate.

Oct 27, 2002 22:56:11

On a tangential note, Dr. Xing's research of purple versus green monkey
trans-sociopolitical impact seems to be stalled, having gained no
ground for several weeks. Today she learned that her lab assistant
never mentioned on his job application that he was colorblind. Oh
well.
```

И снова обратите внимание на RSS-документ версии 1.0:

```
<?xml version="1.0" encoding="UTF-8"?>

<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns="http://purl.org/rss/1.0/"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 xmlns:taxo="http://purl.org/rss/1.0/modules/taxonomy/"
 xmlns:syn="http://purl.org/rss/1.0/modules/syndication/"
>
<channel rdf:about="http://www.jmac.org/linklog/">
<title>Link's Log</title>
<link>http://www.jmac.org/linklog/</link>
<description>Dr. Lance Link's online research journal</description>
<dc:language>en-us</dc:language>
<dc:rights>Copyright 2002 by Dr. Lance Link</dc:rights>
<dc:date>2002-10-27T23:59:15+05:00</dc:date>
<dc:publisher>llink@jmac.org</dc:publisher>
<dc:creator>llink@jmac.org</dc:creator>
<dc:subject>llink</dc:subject>
<syn:updatePeriod>daily</syn:updatePeriod>
<syn:updateFrequency>1</syn:updateFrequency>
<syn:updateBase>2002-03-03T00:00:00+05:00</syn:updateBase>
<items>
```

```
    <rdf:Seq>
      <rdf:li rdf:resource="http://www.jmac.org/linklog?2002-10-27#22:56:11" />
      <rdf:li rdf:resource="http://www.jmac.org/linklog?2002-10-18#19:07:06" />
    </rdf:Seq>
  </items>
  </channel>

  <item rdf:about="http://www.jmac.org/linklog?2002-10-27#22:56:11">
  <title>2002-10-27 22:56:11</title>
  <link>http://www.jmac.org/linklog?2002-10-27#22:56:11</link>
  <description>
  Today I asked lab monkey 45-X how he felt about his recent chess
  victory against Dr. Baker. He responded by biting my kneecap. (The
  monkey did, I mean.) I
  think this could lead to a communications breakthrough. As well as
  painful swelling, which is unfortunate.
  </description>
  </item>

  <item rdf:about="http://www.jmac.org/linklog?2002-10-18#19:07:06">
  <title>2002-10-18 19:07:06</title>
  <link>http://www.jmac.org/linklog?2002-10-18#19:07:06</link>
  <description>
  On a tangential note, Dr. Xing's research of purple versus green monkey
  trans-sociopolitical impact seems to be stalled, having gained no
  ground for several weeks. Today she learned that her lab assistant
  never mentioned on his job application that he was colorblind. Oh well.
  </description>
  </item>

  </rdf:RDF>
```

Обратите внимание на то, что RSS 1.0 использует различные пространства имен, обеспечивающие поддержку метаданных еще до получения доступа к структуре фактического содержимого[1]. У любопытных индивидуумов может возникнуть мысль воспользоваться URI-ссылками, позволяющими выполнить самоидентификацию. Это может понадобиться в случае наличия небольших и «уютных» пространств имен, в которых находится требуемая документация. (Аббревиатура «dc» означает Dublin Core, стандартный набор элементов, применяемых для описания источников документов. Аббревиатура «syn» имеет отношение к пространству имен синдиката, выступающему в качестве подпроекта поддерживаемого пользователя RSS-проекта, состоящего из множества элементов, с помощью которых проверяется частота обновления источника новым содержимым.) Затем документ в целом включается в состав RDF-элемента.

---

[1] Я стараюсь точно указать версии RSS-документов, поскольку RSS-документы версий 0,9 и 0,91 имеют упрощенную структуру. Причем в этом случае не используются пространства имен и RDF-инкапсулированные метаданные. Вместо них применяется простой список из элементов <i tem>, включенный в состав элемента <rss>. Исходя из этих соображений, многие предпочитают использовать версии RSS-документов, предшествующие версии 1.0, поэтому поддерживается совместимость между всеми версиями RSS-программ. Эти возможности присущи модулю XML: :RSS наряду с побочным эффектом, проявляющимся в возможности простого преобразования между различными версиями (в рамках одного исходного документа).

Синтаксический разбор

Порядок использования модуля XML: : RSS с целью чтения существующего документа может показаться вам знакомым (особенно если вы внимательно читали предыдущие главы). Вообще говоря, здесь нет ничего сложного:

```
use XML::RSS;

# Получение файла, передаваемого с помощью
# аргументов пользователя.
my @rss_docs = @ARGV;

# А теперь предполагаем, что все файлы
# находятся на диске...
foreach my $rss_doc (@rss_docs) {

# Сначала создается новый RSS-объект,
# представляющий разбираемый документ.
my $rss = XML::RS5->new;

# Теперь разбирается следующий фрагмент кода.
$rss->parsefile($rss_doc);

# На этом все. Вы свободны в выборе дальнейших действий.
}
```

Наследование модуля XML: : Parser

Если метод parsefileкажется знакомым, это легко объяснимо: аналогичный метод используется «великим предком», модулем XML: : Parser.

Модуль XML: : RSS позволяет непосредственно реализовать преимущество, связанное с применением наследования в XML: : Parser. Последний модуль просто помещается в массив @ISA прежде, чем начнут выполняться соответствующие map-объявления.

Для пользователей, знакомых с объектно-ориентированным программированием на Perl, здесь нет ничего удивительного. Определяется собственный новый метод, причем выполняемый при этом объем работы немного превышает тот, что требуется при вызове модуля SUPER: : new. В этом случае модуль XML: : Parser самостоятельно выполняет инициализацию. Обратите внимание на фрагмент кода из этого модуля, особенно на конструктор new, который вызывается в следующем примере:

```
sub new {
    my $class = shift;
    my $self = $class->SUPER::new(Namespaces     => 1,
                                  NoExpand       => 1,
                                  ParseParamEnt  => 0,
                                  Handler     =>  { Char => \&handle_char,
                                  XMLDecl        => \&handle_dec,
                                  Start          => handle_start})
;
    bless ($self,$class);
    $self->_initialize(@_);
    return $self;
}
```

Обратите внимание: модуль вызывает метод new модуля-предка, используя весьма специфические аргументы. Все они стандартны и хорошо описаны в инструкциях по установке для общедоступного интерфейса XML: : Parser. Передавая эти пара-

метры в распоряжение пользователя, модуль XML: : RSS точно **«знает»,** что он получает в этом случае. Речь идет об объекте анализатора с включенной обработкой пространства имен, но при этом отсутствует расширение либр синтаксический разбор параметров сущностей. Также самостоятельно **определяются** обработчики.

Результат вызова модуля SUPER: : new представляет собой объект XML: : Parser, который вряд ли будет передаваться обратно в распоряжение пользователей модуля. Если передача состоится, это скажется на достигаемой степени абстракции! Таким образом повторно одобряется применение объекта (на этот момент времени рассматриваемого в качестве новой инструкции **$self** для данного класса), использующего корректные (с точки зрения Perl) методы, применяющие два аргумента. В результате возвращаемый объект требует «верности» модулю XML: : RSS, а не модулю XML : : Parser.

Объектная модель

До сих пор мы уже видели, что модуль XML: : RSS не относится к разряду уникальных продуктов с точки зрения устройства объекта анализатора и процесса разбора документов. Теперь же перейдем к рассмотрению области, в которой наиболее ярко проявляются преимущества модуля, — на базе внутренней структуры данных производится создание и применение интерфейсов API, основанных на методах этого модуля.

Большая часть кода модуля XML: : RSS формируется на основе методов-потомков, применяемых для чтения и записи предопределенных позиций в создаваемой структуре. Не используя ничего более сложного, чем несколько хэш-объектов Perl, модуль XML: : RSS формирует карты объектов, которые ожидаются в данном документе. Причем эти карты состоят из вложенных хэш-ссылок с именованными ключами после элементов и возможными атрибутами. Перечисленные объекты являются вложенными, вследствие чего определяется метод их поиска в реальном RSS XML-документе. Модуль определяет одну из подобных карт для каждой версии **обрабатываемого** RSS-документа. Ниже приводится пример простейшей карты, связанной с RSS-документом версии 0.9:

```
my %v0_9_ok_fields = (
    channel =>    {
            title => '',
              description => '',
              link => '',
              }.
    image => {
              title => '',
              url => '',
              link => ''
              >,
    textinput => {
              title =>'' ,
              description => '',
              name => '',
              link => ''
              }•
    items => [],
    num_items => 0,
    version => '',
```

```
        encoding => ''
    );
```

При формировании рассматриваемой модели используются не только хэш-ссылки. Например, ключ «элементы» высшего уровня содержит пустую ссылку на массив. С другой стороны, все конечные значения для ключей являются скалярными. На базе всего перечисленного выше формируются пустые строки. В качестве исключения можно рассматривать элемент num_i terns , который не принадлежит к числу RSS-элементов. Его применение диктуется соображениями удобства, в результате чего достигается компромиссный вариант, проявляющийся в виде структурной элегантности и удобства в применении (возможно, код не поддерживает явное разыменование ссылок на элементы массива, поэтому их значения считываются в скалярном виде).

С другой стороны, при рассмотрении подобного примера можно легко потерять связь с реальностью (в случае, если описываются изменения, а программист не запоминает дату выполнения подобных действий). Подобные вопросы часто имеют отношение к стилю программирования и выходят за рамки нашей книги.

Описываемое распределение имеет свои причины, помимо того, что хэш-объектам может присваиваться что угодно (к этому разряду относится даже undef). Каждый хэш-объект несет двойную нагрузку и может рассматриваться в качестве карты процедур модуля либо шаблона для самих структур. Учитывая это обстоятельство, давайте посмотрим, что происходит при конструировании элемента XML: :Parserс помощью метода класса new для данного модуля.

Ввод: пользователь или файл

Непосредственно после завершения фазы конструирования модуль XML: : RSS может применяться для разбора RSS-документа. Это происходит благодаря возможностям по проведению синтаксического разбора, унаследованным от модуля XML: : Parser. Все, что должен в данном случае сделать пользователь — это вызвать методы объекта parse либо parsefile, и все!

Несмотря на этот примечательный факт, многие из настоящих объектов могут «жить долго[1] и счастливо» без выполнения обработки существующих XML-документов. Зачастую RSS-пользователи хотят, чтобы модуль полностью выполнял работу по созданию документа «с нуля», или компонуя части текста, предназначенные для использования программой. Это как раз тот случай, когда будут использоваться все упомянутые средства обеспечения доступа.

Предположим, что имеется база данных, созданная с применением SQL-команд, содержащая некоторые записи web-журнала, имеющие отношение к RSS-документу. На ваше рассмотрение предоставляется следующий небольшой сценарий:

```
#!/usr/bin/perl

# Включение последних 15 записей web-журнала доктора Пинка
# в RSS-документ версии 1, тут же направляемых в STDOUT.
use warnings;
use strict;
```

[1] На самом деле под «долгой жизнью» подразумевается несколько сотен секунд машинного времени для типичного ПК.

```
use XML::RSS;
use DBIx::Abstract;
my $MAX_ENTRIES = 15;
my ($output_version) = @ARGV;
$output_version ||= '1.0';
unless ($output_version eq '1.0' or $output_version eq '0.9'
                                  or $output_version eq '0.91') {
    die "Usage: $0 [version]\nWhere [version] is an RSS version to output:
0.9, 0 .91, or 1.0\nDefault is 1.0\n";
}

my $dbh = DBIx::Abstract->connect({dbname=>'weblog',
                                  user=>'link',
                                  password=>'dirtyape'})
     or die "Couln't connect to database. \n";

my ($date) = $dbh->select('max(date_added)',
                                   'entry')->fetchrow_array;
my ($time) = $dbh->select('max(time_added)',
                                  'entry')->fetchrow_array;

my $time_zone = "+05:00"; # Часовой пояс. :)
my $rss_time = "${date}T$time$time_zone";
# Базовое время, назначаемое при запуске с blog, для
# информации о синдикате.
my $base_time = "2001-03-03T00:00:00$time_zone";

# Выбран RSS версии 1.0, с помощью которого некоторая
# метаинформация передается в «модули», помещаемые
# в собственные пространства имен, такие как
# 'dc' (аббревиатура для Dublin Core) либо 'syn'
# (аббревиатура для RSS Syndication), к счастью, при
# этом документ нисколько не усложняется, как можно
# видеть в дальнейшем.

my $rss = XML::RSS->new(version=>'1.0', output=>$output_version);

$rss->channel(
               title=>'Dr. Links Weblog',
               link=>'http://www.jmac.org/linklog/',
               description=>"Dr. Link's weblog and online journal",
               dc=> {
                    date=>$rss_time,
                    creator=>'llink@jmac.org',
                    rights=>'Copyright 2002 by Dr. Lance Link',
                    language=>'en-us',
                    }.
               syn=>{
                    updatePeriod=>'daily',
                    updateFrequency=>1,
                    updateBase=>$base_time,
                    },
               );

$dbh->query("select * from entry order by id desc limit $MAX_ENTRIES");
while (my Sentry = $dbh->fetchrow_hashref) {
     # замена сомнительных XML-символов на сущности.
```

```
        $$entry{entry} =~ s/&/&/g;
        $$entry{entry} =~ s/</&lt;/g;
        $$entry{entry} =~ s/'/&apos;/g;
        $$entry{entry} =~ s/"/&quot;/g;
        $rss->add_item(
             title=>"$$entry{date_added} $$entry{time_added}",
         link=>"http://www.jmac.org/weblog?
        $$entry{date_added}#$$entry{time_added}",
            description=>$$entry {entry}",
                      );
}

# Результаты просто направляются в стандартный поток вывода. :)
print $rss->as_string;
```

Видели ли вы здесь признаки XML-кода? Лично нам не удалось их узреть. Да, здесь применялись регулярные выражения, но фактически производилось чтение данных из базы данных, а затем помещение найденного в объект путем вызова нескольких методов (точнее, речь идет о вызове единственного метода add_item, осуществляемого в цикле). Эти вызовы и их единственные аргументы принимаются с помощью хэш-объекта, созданного на основе нескольких простых строк. При написании этой программы преследовалась определенная цель — использование при создании web-журнала всех преимуществ, предлагаемых RSS, причем при создании нашего файла XML-код не применялся.

Импровизированный вывод

Таким образом, модуль XML: :RSS не может воспользоваться модулями-помощниками, обеспечивающими генерирование XML-кода (например, XML: :Writer) с целью генерирования соответствующего вывода. В этом случае просто создается одно длинное скалярное значение, основываясь на котором хэш-карта приобретает вид кода. Выполнение последнего регулируется с помощью обычных блоков if, else и elsif. Причем каждый из этих блоков тяготеет к использованию оператора самоконкатенации, . ='. Если же вы хотите приступить к созданию собственных модулей, генерирующих XML-код, попытайтесь воспользоваться следующим подходом. Создайте документ в памяти, состоящий из литералов, затем выведите его на печать, воспользовавшись файловым дескриптором. В результате вы сможете уменьшить издержки и получить дополнительный уровень контроля, хотя сам процесс станет менее безопасным. Удостоверьтесь в том, что вывод был проверен и признан формально корректным. (Если применяется комбинированный инструмент анализатор/генератор, наподобие XML: :RSS, попытайтесь выполнить синтаксический разбор результатов, выводимых модулем, и убедитесь в том, что получается то, что ожидалось.)

Инструменты XML-программирования

А теперь рассмотрим программное обеспечение, функции которого отличаются от того, что было описано ранее. Вместо того чтобы обрабатывать XML-документы с применением Perl-методики, используются XML-стандарты. В этом случае обеспечивается решение задач, не требующих явного применения XML. В настоящее время многие ведущие специалисты, фигурирующие в списке рассылки perl-xml,

заняты поиском мини-платформы, пригодной для универсальной обработки данных средствами Perl. Причем в качестве основы в данном случае используется SAX. В результате этого исследования появляются некоторые интересные (и полезные) примеры. К их числу можно отнести модули XML::SAXDriver::Excel и XML::SAXDriver::CSV, разработанные Ильей Стериным (Ilya Sterin), и модуль XML::Generator::DBI, разработанный Мэттом Сержантом (Matt Sergeant). Эти три модуля могут получать данные в формате файлов Microsoft Excel, содержащих разделенные запятыми значения, а также в формате баз данных SQL. Роль оболочки в данном случае играют SAX API (Некоторые из этих компонентов рассматривались в главе 5, поэтому любой программист вправе рассчитывать на то, что новый формат столь же хорош и легко управляем, как и другие, ранее рассматриваемые XML-документы.)

Обратимся к подробному рассмотрению одного из этих инструментов, представляющему особый интерес в свете современных концепций программирования.

Модуль XML::Generator::DBI

Модуль XML::Generator::DBI может служить превосходным примером так называемого *склеивающего модуля*. Основное назначение этой простой программы заключается в обеспечении взаимодействия между двумя существующими (но не полностью связанными) частями программного обеспечения. В этом случае, при конструировании объекта из данного класса, можно встраивать дополнительные объекты: дескриптор базы данных DBI и объект обработчика, «говорящий» на языке SAX.

Модуль XML::Generator::DBI не располагает сведениями относительно происхождения объектов. Здесь реализованы доверительные отношения, основанные на отклике в случае вызова стандартных методов и соответствующих им семейств методов (DBI, SAX или SAX2). Затем на базе объекта XML::Generator::DBI вызывается метод execute. При этом используется стандартная SQL-конструкция, как и в случае с дескриптором базы данных, созданной на основе DBI.

В следующем примере демонстрируется практическое применение модуля. Используемый SAX-обработчик — экземпляр модуля XML::Handler::YAWriter, разработанного Майклом Коэном (Michael Koehne). Этот модуль легко настраивается, причем в результате его применения SAX-события преобразуются в текстовый вывод. Благодаря применению этой программы можно, к примеру, SQL-таблицу, содержащую перечень компакт-дисков, преобразовать в формально корректный XML-документ и вывести его на стандартное устройство печати:

```
#!/usr/bin/perl

use warnings;
use strict;

use XML::Generator::DBI;
use XML::Handler::YAWriter;

use DBI;

my $ya = XML::Handler::YAWriter->new(AsFile=> "-");
my $dbh = DBI->connect("dbi:mysql:dbname=test", "jmac", "");
my $generator = XML::Generator::DBI->new(
```

```
                                Handler => $ya,
                                dbh => $dbh
                                );
my $sql = "select * from cds";

$generator->execute($sql);
```

Ниже приводится соответствующий результат:

```
<?xml version="1.0" encoding="UTF-8"?><database>
    <select query="select * from cds">
        <row>
            <id>1</id>
            <artist>Donald and the Knuths</artist>
            <title>Greatest Hits Vol. 3.14159</title>
            <genre>Rock</genre>
        </row>
        <row>
            <id>2</id>
            <artist>The Hypnocrats</artist>
            <title>Cybernetic Grandmother Attack</title>
            <genre>Electronic</genre>
        </row>
        <row>
            <id>3</id>
            <artist>The Sam Handwich Quartet</artist>
            <title>Handwich a la Yogurt</title>
            <genre>Jazz</genre>
        </row>
        </select>
</database>
```

Настоящий пример не представляет особого интереса, хотя и хорошо выглядит при печати. В этом случае не использовался модуль YAWriter. В системе можно применять любой из имеющихся SAX-обработчиков Perl-пакета, включая те из них, которые были написаны пользователем. Последние могут задействоваться при создании новых объектов XML: :Generator: : DBI. В результате применения описанного ранее примера таблицы базы данных при вызове метода execute объекта $genenerator создается эффект, аналогичный итогам синтаксического разбора предыдущего XML-документа (различается лишь количество пробелов, включаемых YAWriter с целью улучшения восприятия текста пользователем). Причем эта закономерность может проявляться даже в том случае, если фактически исходный код не относится к разряду XML-документов, а просто является таблицей базы данных.

Размышления по поводу DBI и SAX

А теперь сделаем небольшое отступление в сторону методик, основанных на применении DBI и SAX. При этом доступны возможности общего характера, обеспечивающие управление данными.

Основная причина популярности Perl DBI, используемого в качестве интерфейса базы данных Perl, заключается в особенностях архитектуры. При инсталляции DBI, необходимы два отдельных компонента: DBI.pm включает код DBI API, а также всю необходимую документацию, но в этом случае вы не можете управлять базой данных Perl. Полный спектр возможностей обеспечивается при наличии, как минимум, одного DBD-модуля, совместимого с типом применяемой базы данных. В сети

CPAN можно найти множество подходящих модулей: DBD: : MySQL, DBD: : Oracle и DBD: : Pg (для базы данных Postgres). Если программист организует взаимодействие с одним DBI-модулем путем отправки ему SQL-отчетов и получения соответствующих результатов, с реальной базой данных взаимодействует соответствующий DBD-модуль. Благодаря применению DBD-модуля абстрактные DBI-методы преобразуются в высшей степени специфичные и зависящие от применяемой платформы команды базы данных. Причем рабочий уровень в данном случае находится существенно ниже уровня, на котором работают DBI-пользователи. В результате, любая Perl-программа, использующая возможности DBI, сможет обрабатывать любую базу данных (в случае наличия подходящего DBD-драйвера[1]).

В мире Perl и XML наблюдается быстрый прогресс, причем начало «взрывообразного» развития датируется 2001 годом (время возникновения SAX-драйверов, упомянутых в начале главы). Логическим результатом этого прогресса стала разработка модуля XML : : SAX, SAX2-реализация которого функционирует подобно DBI. Пользователю достаточно указать желаемый SAX-анализатор, дополнительно определить SAX-свойства, присущие программе, после чего осуществляется поиск наилучшего рабочего инструмента в системе, создание рабочего экземпляра инструмента и его передача в распоряжение пользователя. Затем можно подключиться к пакету выбранного SAX-обработчика (как и в случае с модулем XML : :Generator: : DBI) и все готово.

Благодаря PerlSAX-обработчикам можно использовать стандартный интерфейс с целью выборки данных из любого мыслимого источника данных, в то время как DBD-драйверы обеспечивают стандартный интерфейс для выборки данных из баз данных. Как и в случае с DBI, требуется всего лишь неустрашимый хакер, который сможет разобраться в тонкостях реализации применяемого формата данных, после чего другие Perl-программисты смогут воспользоваться стандартными API с целью обработки «чужеродного» формата. Для этого им потребуется разобраться в деталях реализации SAX.

Модуль SOAP::Lite

И напоследок мы приступим к рассмотрению категории программ Perl и XML, которые настолько далеки от тематики, что, может быть, и не стоило упоминать о них, если бы не некоторые примечательные присущие им свойства. В этой категории описываются модули и расширения, которые подобны модулям-помощникам из класса XML: : RSS. Благодаря их применению можно работать с разнообразными XML-документами. Причем в этом случае программисты «изолируются» от исходного потока данных, «нашпигованного» элементами. Количество уровней абстракции в этом случае таково, что присутствие XML-кода вообще не ощущается.

Например, если требуется написать программу, использующую возможности протоколов SOAP либо XML-RPC по обработке удаленного кода, вряд ли вы сможете это сделать лучше специализированных программных модулей. В этом слу-

[1] Предположим, например, что программист не позаботился о том, чтобы под рукой была программа, обеспечивающая уникальные запросы для базы данных. В этом случае запрос типа $sth->query('select lasr_insert_id() from foo') может хорошо выполняться по отношению к базе данных MySQL, но мало подходит при работе с базами данных Postgress.

чае в распоряжение программиста предоставляется некая «скатерть-самобранка», выполняющая всю черновую работу по созданию программного кода! (По желанию пользователя, хороший модуль обеспечивает доступ к исходному XML-коду.)

Протокол простого доступа к объектам (Simple Object Access Protocol, SOAP) предоставляет в распоряжение программиста всю мощь объектно-ориентированных web-служб[1]. При этом он может конструировать и использовать объекты, для которых с помощью URI-ссылки устанавливается соответствие с определениями классов. В этом случае даже не требуется знать применяемый язык программирования, поскольку протокол преобразует методы объектов в универсальные API, основанные на XML-коде. До тех пор, пока класс где-либо документирован, дополнительные детали доступных классов и методов объекта можно исключить в том случае, если класс представляет собой просто другой файл на локальном жестком диске. Причем то, что он фактически существует на удаленном компьютере, не имеет ни малейшего значения.

К этому времени можно легко забыть о том, что осуществляется работа с XML-кодом. По крайней мере, при работе с RSS имена методов объектных API более или менее соответствуют аналогичным именам в результирующем выходном документе. В этом случае вряд ли вы захотите видеть документы, предназначенные исключительно для компьютера. Это нежелание сродни неприязни к числовым кодам, представляющим коды клавиш, которые пересылаются центральному процессору компьютера.

Имена, определенные в `SOAP::Lite`, соответствуют планируемому объему работы и нисколько не отражают ваше собственное мнение. После установки этого модуля в системе в распоряжение программиста предоставляется большой перечень доступных Perl-пакетов, многие из которых определяют большое количество транспортных стилей[2]. Модуль `mod_perl1` облегчает поддержку web-служб средствами SOAP, а также включает полный пакет, предлагающий в распоряжение пользователя соответствующую документацию и примеры. Этот модуль может применяться в аналогичных целях совместно с набором модулей, поддерживающих подобные API для модуля XML-RPC, напоминающего SOAP, но не поддерживающего объектно-ориентированные свойства. Модуль `SOAP::Lite` может служить своего рода источником вдохновения для Perl-программистов, успехи которых в программировании web-служб сравнимы с возможностями CGI.pm в сфере программирования динамических web-узлов.

А теперь засучите рукава и приготовьтесь к началу работы с SOAP.

Первый пример: преобразователь температур

В каждой книге, посвященной программированию, обязательно имеется пример программного кода, выполняющего преобразование значений температур, не так ли?

---

[1] Несмотря на название, web-службы не связаны непосредственно с World Wide Web. На самом деле web-служба — часть программы, выполняющая прослушивание порта, на который указывает URI. После приема запроса формируется отклик, имеющий смысл для запрашивающей сущности. В качестве наиболее общего примера Web-службы можно рассматривать web-сервер, поддерживающий передачу данных с применением протокола HTTP. Концепция более современных центров рекламы предполагает поддержку постоянного доступа к объектам и процедурам (в результате программист может использовать программный код, хранящийся на удаленных серверах, с целью его интеграции с локальным программным обеспечением).

[2] Как правило, в этом случае SOAP-объекты используют HTTP, но если вы захотите воспользоваться исходным протоколом TCP, SMTP либо даже Jabber, особых проблем не возникает.

Мы не будем нарушать эту традицию. В рассматриваемом примере, позаимствованном из раздела SYNOPSIS документации по модулю SOAP::Lite, была создана программа, основная функция которой, f2c, находится на web-узле по адресу http://services.soaplite.com/Temperatures.

```
use SOAP::Lite;
print SOAP::Lite
  ->uri('http://www.soaplite.com/Temperatures')
  ->proxy('http://services.soaplite.com/temper.cgi')
  -> f2c(32)
  -> result;
```

При выполнении этой программы в качестве Perl-сценария (на компьютере с корректно установленным модулем SOAP::Lite) получаем корректный результат: 0.

Второй пример: механизм поиска индексов ISBN

На этот раз рассмотрим небольшой модуль, который можно найти на одном из персональных web-серверов автора. Этот модуль носит ярко выраженный объектно-ориентированный характер. В качестве входных данных берется ISBN-номер, на основе которого формируется XML-код Dublin Core практически для каждой книги, которая может ему соответствовать:

```
my($isbn_number)=@ARGV;
use SOAP::Lite +autodispatch=>
    uri=>'http://www.jmac.org/ISBN',
    proxy=>'http://www.jmac.org/projects/bookdb/isbn/lookup.pl';
    my $isbn_obj = ISBN->new;
# Метод 'get_dc' выбирает информацию Dublin Core.
my $result = $isbn_obj->get_dc($isbn_number);
```

Весь секрет заключается в том, что модуль, размещенный на хост-компьютере, ISBN.pm, не является особо необычным. Этот достаточно простой Perl-модуль может применяться обычным образом в случае, если требуется располагать локальной копией. Другими словами, можно получить те же самые результаты путем регистрации на компьютере и запуска на выполнение следующей программы:

```
my ($isbn_number) = @ARGV;
use ISBN; # ЭТОТ КОД заменяет 'use SOAP::Lite' строкой
my $isbn_obj = ISBN->new;
# Метод 'get_dc' выбирает информацию Dublin Core.
my $result = $isbn_obj->get_dc($isbn_number);
```

Вызов модуля SOAP::Lite и использование некоторых дополнительных методик облегчает обзор удаленного компьютера, «прослушивающего» запросы в стиле SOAP. В этом случае не требуется копировать Perl-модуль на клиентский компьютер с целью применения соответствующих интерфейсов API. В конечном счете, мы приходим к странному повторно реализованному модулю Java, о котором вы, возможно, ничего и не знаете, несмотря на подобие применяемых интерфейсов. В мире web-служб, не зависимых от языков, все имеет значение.

Где же тут XML-код? Давайте посмотрим, что происходит в том случае, если конструктор класса ISBN вызывается после активизации опции outputxml модуля SOAP::Lite:

```
my ($isbn_number) = @ARGV;
use SOAP::Lite +autodispatch=>
```

```
        uri=>'http://www.jmac.org/ISBN', outputxml=>1,
        proxy=>'http://www.jmac.org/projects/bookdb/isbn/lookup.pl';
    my $isbn_xml = ISBN->new;
    print "$isbn_xml\n";
```

И вот мы возвращаемся к следующему коду:


```
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://
www.w3.org/1999/XMLSchema-instance" xmlns:xsd="http://www.w3.org/1999/
XMLSchema" xmlns:namesp1="http://www.jmac.org/ISBN"><SOAP-
ENV:Body><namesp2:newResponse xmlns:namesp2="http://www.jmac.org/
ISBN"><ISBN xsi:type="namesp1:ISBN"/></namesp2:newResponse></SOAP-
ENV:Body></SOAP-ENV:Envelope>
```


Второй фрагмент кода листинга приводит к прекращению выполнения, поскольку возвращается скалярное значение, содержащее большой объем XML-кода (который выводится на печать) вместо объекта, принадлежащего семейству классов SOAP::Lite. В этом случае невозможно продолжение вызовов методов. Эта проблема может быть устранена путем использования класса SOAP::Deserializer, возвращающего XML-код SOAP XML обратно в распоряжение объектов:

```
# Продолжение предыдущего фрагмента кода.
my $deserial = SOAP::Deserializer->new;
my $isbn_obj = $deserial->deserialize($isbn_xml);
# Можно продолжать, как и в первом примере.
```

Небольшой объем дополнительной работы может способствовать «увязанию в дебрях» исходного кода XML, равно как среди «черных ящиков» объектов SOAP::Lite. Как и следовало ожидать, рассматриваемое свойство имеет отношение не только к примерам данной книги. Предоставление исходного XML-кода в распоряжение пользователя влечет за собой появление самых разных интересных ошибок, причиной которых может стать «человеческий фактор».

В то время как магия Perl-модуля SOAP::Lite проявляется различными путями, сам SOAP является просто протоколом, который наравне с причудливыми пространствами имен, элементами и атрибутами, присущими XML и генерируемыми рассматриваемым модулем, удовлетворяют требованиям SOAP-спецификации[1]. Благодаря подобной совместимости возможно применение хитроумного плана по отношению к SOAP-приложениям. В этом случае модуль SOAP::Lite может выполнять свои традиционные функции. Затем эти функции переходят к вашей программе, которая захватывает исходный XML-код, выполняет по отношению к этому коду некие странные и непонятные на первый взгляд операции (код может разбираться с помощью любого ранее рассмотренного метода), а затем возвращает контроль обратно модулю SOAP::Lite.

Общеизвестно, что при выполнении большинства операций с модулем SOAP::Lite не требуется четко указывать границы сведений относительно обработки XML-кода в Perl. Большинство приложений заранее снабжены требуемым набором функциональных свойств. Если вы хотите получить реальную выгоду из этого обстоятельства, дерзайте. Знание — сила!

[1] Описание версии 1.2 содержится на web-узле по адресу http://www.w3.org/TR/soap12-part1/.

10 Стратегии программирования

В этой главе повторно затронуты темы большинства разделов книги. Мы возвратимся к рассмотрению многих вопросов XML-обработки в Perl, представленных в главе 3, однако в контексте всего того материала, с которым вы ознакомились в предыдущих главах. Цель состоит в проведении завершающего обзора Perl и XML, с его стратегиями и понятиями.

Пространства имен Perl и XML

Пространства имен XML использовались с тех пор, как этот термин был впервые упомянут в главе 2. Многие XML-приложения, например, XSLT, построены на том, что все их элементы используют определенное пространство имен. Решающим фактором при этом, как правило, служит то, в какой степени является комбинированным приложение при традиционном применении: выполняется ли оно само по себе, то есть один документ на одно приложение, или смешиваются другие виды XML-обработки документов?

Например, XML-документ DocBook не является в высокой степени комбинированным. Экземпляр DocBook - это практически всегда XML-документ, определяющий книгу или статью, а все элементы этого документа неявно связаны с некоторым другим пространством имен. Описание этого пространства имен можно найти в официальной документации по DocBook[1]. Однако, используя этот документ, можно натолкнуться на ряд элементов MathML, «приютившихся» среди элементов DocBook и «впитывающих» в себя содержимое последних.

Этот вид элементов полезен по двум причинам: во-первых, хотя элементы DocBook разрабатывались с целью охвата всего многообразия форматов и стилей,

[1] За дополнительными сведениями обратитесь к web-узлу http://www.docbook.org либо к книге DocBook: The Definitive Guide издательства O'Reilly.

которые можно найти в технической документации[1], с их помощью, например, невозможно адекватно описывать математические уравнения. (В состав этого документа действительно входят элементы <equation>,Но они часто используются для описания природы графических элементов.) Добавив возможности MathML, можно использовать все теги, определенные этой спецификацией языка разметки в DocBook-документе и надежно «упрятанные» в пространстве имен. (Поскольку MathML и DocBook могут использоваться совместно, DocBook DTD разрешает пользователю включать «модуль MathML», который добавляет в программу элемент `<mml:math>`. В этой программе каждый элемент обрабатывается с помощью своего документа DTD для MathML, который импортирует этот модуль (вместе с основным DTD DocBook) во все DTD-пространство при проверке достоверности.))

Во-вторых, возможно, более интересная причина с позиций анализатора состоит в том, что теги, существующие в данном пространстве имен, работают как «посольства»; пока вы находитесь на их территории (другими словами, в их области действия) все законы и правила этой страны применяются к вам, несмотря на размещение посольства на территории Другого государства. Пространства имен XML также похожи на пространства имен Perl, которые позволяют применять переменные, подпрограммы и другие идентификаторы, заложенные в модуле `Some::Other::Package`, хотя они могут быть не определены по умолчанию в основном пакете (или в каком-либо другом пакете, с которым вы работаете).

Другими словами, наличие пространства имен часто указываетна то, что из данного приложения запускается второе, самостоятельное XML-приложение. Поэтому, если известно, что какой-либо процессор написан для обработки некоторого вида XML-приложений и в нем, возможно, будет использоваться определенное пространство имен, можно значительно облегчить себе работу, передав управление другому Perl-модулю, который может выполнять обработку вместо этого второго XML-приложения.

Например, на диске вашего компьютера находится XML-файл, документ которого содержит список обезьян, живущих в доме. Большая часть этого файла содержит элементы в соответствии с вашим замыслом, но, поскольку вы одновременно и хитрый, и ленивый, документ также использует язык разметки обезьян как стандартный способ описания обезьян средствами XML. Это связано с тем, что он был разработан с учетом применения в более крупных документах, в которых определяется собственное пространство имен:

```
    <?xml version="1.0">
<monkey-list>
    <monkey>
        <description xmlns:mm="http://www.jmac.org/projects/monkeys/mm/">
            <mm:monkey> <!-- начало секции monkey -->
                <mm:name>Virtram</mm:name>
                <mm:color>teal</mm:color>
                <mm:favorite-foods>
                <mm:food>Banana</mm:food> <mm:food>Walnut</mm:food>
                </mm:favorite-foods>
                <mm:personality-matrix>
                F6 30 00 0A 1B E7 9C 20
                </mm:personality-matrix>
```

[1] Иногда поиск сведений в подобной документации весьма затруднен; этим обусловлено существование «облегченных» вариантов DocBook.

```
            </mm:monkey>
        </description>
        <location>Living Room</location>
        <job>Scarecrow</job>
    </monkey>
    <!-- Здесь будут помещены другие обезьяны -->
</monkey-list>
```

URI-ИДЕНТИФИКАТОРЫ

Многие XML-технологии, например, пространства имен XML, SAX2 и SOAP, основаны на URI как на уникальных строках-идентификаторах, которые дифференцируют возможности и свойства для предотвращения идеологических конфликтов. Любой процессор, который считывает URI, может быть абсолютно уверен в том, что он ссылается на технологию, подразумеваемую автором. Используемые **URI-ссылки** часто выглядят как URL, как правило, в формате http://. В этом случае предполагается, что такая строка при отображении в окне web-браузера приведет к какому-либо результату. Однако иногда единственным результатом является неутешительный ответ HTTP 404. URI-ссылки, в отличие от URL, не должны указывать на фактический ресурс, они лишь должны быть глобально уникальными.

Разработчики, которым необходимо определить новую **URI-ссылку,** часто основывают ее на URL разработанного ими ранее web-узла. Например, если ранее был спроектирован web-узел http://**www.greenmonkey-markup.com/~jmac,** то, основываясь на нем, можно определить такой URI, как **http://www.greenmonkey-markup/~jmac/monkeyml/.** Даже не получив ответ, все же можно гарантировать, что никто другой никогда не будет использовать этот URI. Однако дотошные разработчики заботятся о том, чтобы записать в документацию по **URI-предпочтениям** какие-либо сведения об использованной технологии.

Другое популярное решение состоит в использовании такой службы, как **http://purl.org** (не имеющей отношения к Perl). С ее помощью добавляется дополнительный уровень преобразования между URI, используемым в качестве пространства имен, и адресом, содержащимся в его документации; этот уровень позволяет при желании изменить адрес, в то время как URI остается **постоянным.**

Иногда **URI** на самом деле содержит информацию (помимо того что просто является уникальным). Например, многие процессоры XML-приложений достаточно обоснованно придерживаются URI, использующихся для объявления пространств имен. Так, для процессоров XSLT, как правило, важно не столько, что у всех XSLT-элементов таблицы стилей есть обычный префикс `xsl:`, сколько то, какой URI этим префиксом ограничивается в соответствующем присоединенном атрибуте `xmlns:`. Знание того, какой URI ограничивается префиксом, указывает процессору на то, что требуется использовать, например, последнюю версию XSLT группы **W3C,** а не версию, предваряющую 1.0, адаптированную к самому последнему процессору (который имеет свое пространство имен).

Модуль `XML::NamespaceSupport` Робина Берджона (Robin **Berjon),** доступный в CPAN, может помочь в обработке XML-документов, которые используют пространства имен, и управляет преобразованиями префикса в URI.

К счастью, в системе существует Perl-модуль `XML::MonkeyML`, который может разобрать документ MonkeyML, образовав объект. Этот модуль полезен, поскольку класс `XML::MonkeyML` включает программу для обработки индивидуальности MonkeyML - элемента матрицы, в котором сведена воедино вся индивидуальность обезьяны вплоть до короткого шестнадцатеричного кода. Напишем программу, которая предсказывает, как все обезьяны будут реагировать в данной ситуации:

```
#!/usr/bin/perl
# Эта программа выполняет действия по указанию в командной
# строке и применяет их к каждой обезьяне из списка
# обезьян XML-документа (причем содержащий документ
# файл также указывается в
# командной строке).
```

```
use warnings;
use strict;

use XML::LibXML;
use XML::MonkeyML;

my ($filename, $action) = @ARGV;

unless (defined (Sfilename) and defined (Saction)) {
     die "Usage: $0 monkey-list-file action\n";
}

my Sparser = XML::LibXML->new;
my $doc = $parser->parse_file($filename);

# Получить все элементы обезьян.
my@monkey_nodes=$parser->documentElement->findNodes("//monkey/description/
mm:monkey");
foreach (@monkey_nodes) {
     my Smonkeyml = XML::MonkeyML->parse_string($_->toString);
     my Sname = $monkeyml->name . " the " . $monkeyml->color . " monkey";
     print "Sname would react in the following fashion:\n";
     # Метод объекта MonkeyML 'action' получает на английском языке
     # описание действия, совершаемого над этой обезьяной,
     # и возвращает фразу, описывающую ответное действие обезьяны.
     print $monkeyml->action($action); print "\n";
}
```

Результат выглядит следующим образом:

```
$ ./money_action.pl monkeys.xml "Give it a banana"

Virtram the teal monkey would react in the following fashion:
Take the banana. Eat it. Say "Ook".
```

Потакая лени, посмотрим, как программист может создать модуль-помощник вида XML::MonkeyML.

Создание подклассов

При создании Perl-модулей XML-крекинга есть другой способ «полентяйничать» — залезть на плечи гигантов посредством создания подклассов общих XML-анализаторов, как быстрого способа построить специфические для приложения модули.

Не требуется наследовать объекты; наименее сложный способ выполнить такой вид работы включает создание объекта анализатора обычным способом, прикрепляя его там, где удобно, и, изменяя всякий раз, когда требуется сделать что-либо на XML. Рассмотрим некоторую фиктивную программу:

```
package XML::MyThingy;

use strict; use warnings;

use XML::SomeSortOfParser;
sub new {
     # Старый конструктор.
     my Sinvocant = shift;
     my Sself = {};
     if (ref($invocant)) {
          bless (Sself, ref($invocant));
     } else {
          bless (Sself, Sinvocant);
```

```
    }
    # Теперь создаем XML-анализатор...
    my Sparser = XML::SomeSortOfParser->new
         or die "Oh no, I couldn't make an XML parser. How very sad.";
    # ...и прикрепляем его к этому объекту для
    # дальнейших ссылок.
    $self->{xml} = Sparser;
    return $self;
}
sub parse_file {
    # Только передадим запрос пользователя
    # объекту анализатора (у которого,
    # как оказалось, есть метод, называющийся parse_file)...
    my $self = shift;
    my $result = $self->{xml}->parse_file;
    # То, что произойдет сейчас, зависит от того, что
    # делает объект
    # XML::SomeSortOfParser, когда он анализирует файл.
    # Предположим, что он модифицируется и возвращает
    # код успешного
    # выполнения, поэтому продолжим выполнять только
    # что модифицированный
    # объект с ключом 'xml' и возвратим код.
    return Sresult;
}
```

Тем не менее, выбранный из подкласса анализатор имеет некоторые достоинства. Во-первых, он передает модулю тот же базовый API пользователя, что и модуль в запросе, включая все методы для анализа, полезные для ленивых программистов, особенно если созданный модуль является модулем-помощником XML-приложения. Во-вторых, если используется анализатор, основанный на деревьях, то можно воспользоваться прежними достижениями и расширить представление этого анализатора о структуре данных анализируемого документа, и затем изменить его, чтобы он как можно лучше служил подловатой цели крекинга, пока выполняет специальные действия. Такой шаг возможен благодаря переопределению и наследованию классов Perl.

Пример создания подкласса XML::ComicsML

В этом примере используется воображаемый модуль MonkeyML, отходящий в сторону от «суровой реальности» ComicsML, — языка разметки, применяемого для описания интерактивных комиксов[1]. При этом совместно используются многие возможности и принципы RSS. В результате, помимо всего прочего, поддерживается стандартный путь для комиксов при распределении web-информации, поэтому модуль-помощник ComicsML может быть удобным средством для любого Perl-хакера, желающего создавать программы, обрабатывающие web-комиксы.

Продолжим путь DOM для этого примера и воспользуемся XML::LibXML в качестве внутреннего механизма выбора. Этот анализатор DOM-совместимый и достаточно быстродействующий. В данном случае наша цель состояла в том, чтобы создать полностью объектно-ориентированный интерфейс API, применяемый для обработки ComicsML-документов и всех их основных дочерних элементов:

[1] Дополнительные сведения можно найти на web-узле http://comicsml.jmac.org/.

```
use XML::ComicsML;

# Разбор существующего файла ComicsML.
my Sparser = XML::ComicsML::Parser->new;
my $comic = $parser->parsefile('my_comic.xml');

my $title = $comic->title;
print "The title of this comic is $title\n";
my @strips = $comic->strips;

print "It has ".scalar(@strips)." strips associated with it.\n";
```

Не испытывая особых затруднений, приступим к программированию.

```
package XML::ComicsML;
# Модуль-помощник, предназначенный для разбора
# и формирования ComicsML-документов.
use XML::LibXML;

use base qw(XML::LibXML);

# Разбор.

# Мы перехватили выводимые данные для всех методов
# разбора XML::LibXML, а затем продолжим переопределять
# выбранные вершины в собственные небольшие по
# размеру классы.
sub parse_file {
     # обычный разбор, но после этого переопределяется
     # и возвращается корневой элемент.
     my Sself = shift;
     my $doc = $self->SUPER::parse_file(@_);
     my $root = $doc->documentElement;
     return $self->rebless($root);
}
sub parse_string {
     # Обычный разбор, но после этого переопределяется
     # и возвращается корневой элемент.
     my Sself = shift;
     my $doc = $self->SUPER::parse_string(@_);
     my $root = $doc->documentElement;
     return $self->rebless($root);
}
```

ЧТО именно здесь сделано? До настоящего времени объявлен пакет, являющийся потомком XML: : LibXML (с целью использования базового псевдокомментария), но затем появились собственные версии методов анализа деревьев. Тем не менее, все они выполняют те же действия: вызывают свои методы XML: : LibXML с теми же именами, сохраняют корневой элемент возвращаемого объекта дерева документа и затем передают его внутренним методам:

```
sub rebless {
     # получим некоторый вид XML::LibXML::Node
     #(или его подкласс) и, основываясь на его имени,
     # переопределим его в один из собственных
     # классов ComicsML.
     my Sself = shift;
     my ($node) = @_;

     # Определяем, какие типы элементов интересны
     #(используется хэш для упрощения поиска.)
     my %interesting_elements = (comic=>.l,
```

```
                                   person=>1,
                                   panel=>l,
                                   panel-desc=>1,
                                   line=>l,
                                   strip=>l,
                               . );
    # Отбросим эту вершину, если это не Element либо
    # что-то интересное.
    # Иначе продолжить работу.
        my $пате = $node->getName;
        return $node unless ( (ref($node) eq 'XML::LibXML::Element')
            and (exists($interesting_elements{$name})) );
        # ЭТО интересный элемент! Сообразим, какой класс
        # ему определить и сделаем это.
        my $class_name = $self->element2class($name);
        bless ($node, $class_name);
    return $node;
>
sub element2class {
    # Изменим имя XML-элемента в соответствии с именем класса.
    my $self = shift;
    my ($class_name) = @_;
    $class_name = ucfirst($class_name);
    $class_name =~ s/-(.?)/uc($1)/e;
    $class_name = "XML::ComicsML::$class_name";
}
```

Метод rebless получает вершину элемента, считывает ее имя и проверяет, содержится ли она в жестко запрограммированном списке, содержащем «интересные» имена элементов. Если она появляется в этом списке, метод выбирает для нее имя класса (с помощью «глупого» метода element2class) и переопределяет ее в этом классе.

Такое поведение может показаться иррациональным, если не учитывать тот факт, что объекты XML: :LibXML не особенно устойчивые. Виной тому является способ, с помощью которого они связываются с низкоуровневыми С-подобными структурами, образующими «фундамент» Perl. Если получить список объектов, представляющий нескольких потомков вершины, и затем вновь затребовать его же, могут появиться другие Perl-объекты. Несмотря на это, оба списка могут оказаться функциональными (выступая в качестве API для аналогичных структур в дереве С-библиотеки). Благодаря недостатку устойчивости обеспечивается обход всего дерева в процессе разбора документа. При этом определяются и вызываются «интересные» элементы в специальных классах ComicsML.

Чтобы действовать в соответствии с таким поведением, необходимо выполнить некоторую «черновую работу», которая заключается в преобразовании объектов Element. Эти объекты преобразуются модулем XML : : LibXML в собственные виды объектов там, где это возможно. Основное достоинство этого способа (помимо радости по поводу того, что результатам чей-то работы присваивается имя нашего класса), заключается в том, что можно вызывать эти переопределенные объекты. Наконец, можно определять эти классы.

Во-первых, обратите внимание на метод AUTOLOAD, который присутствует в виртуальном базовом классе XML: :ComicsML: : Element. На базе этого метода наследуются все ваши «реальные» классы элементов. При просмотре кода бросаются в глаза все базовые элементы-потомки и средства доступа к атрибутам классов эле-

мента, вызванные при инициализации неопределенного метода (в качестве примера может рассматриваться метод AUTOLOAD). В этом случае сначала проверяется, есть ли этот метод в жестко запрограммированном списке разрешенных элементов и атрибутов потомка класса (доступны посредством методов `element()` и `attributeO` соответственно); в случае отсутствия таковых, вносится в список метод-конструктор либо метод-деструктор (в зависимости от используемых имен, `add_foo` или `remove_foo`, соответственно):

```
package XML::ComicsML::Element;
# Абстрактный класс для всех видов вершин ComicsML.
use base qw(XML::LibXML::Element);
use vars qw($AUTOLOAD @elements @attributes);

sub AUTOLOAD {
    my $self = shift;
    my $name = $AUTOLOAD;
    $name =~ s/^.*::(.*)$/$1/;
    my @elements = $self->elements;
    my @attributes = $self->attributes;
    if (grep (/^$name$/, ©elements)) {

        # Это средство доступа элемента.
        if (my $new_value = $_[0]) {
        # Установим значение, переписав значение
        # любого текущего элемента этого типа
        my $new_node = XML::LibXML::Element->new($name);
        my $hew_text = XML::LibXML::Text->new($new_value);
        $new_node->appendChild($new_text);
        my @kids = $new_node->childNodes;
        if (my ($existing_node) = $self->findnodes("./$name")) {
            $self->replaceChild($new_node, $existing_node);
        } else {
            $self->appendChild($new_node);
              }
        }

        # Возвратить именованное значение потомка.
        if (my ($existing_node) = $self->f indnodesC ./$name")) {
            return $existing_node->firstChild->getData;
        } else {
            return '';
              }

        } elsif (grep (/^$name$/, ©attributes)) {
    # Это средство доступа атрибута.
    if (my $new_value = $_[0]) {
        # Установить значение для этого атрибута.
        $self->setAttribute($name, $new_value);
    }

        # Возвратить именованное значение атрибута.
        return $self->getAttribute($name) || '';

        # Следующие два метода могут использовать некий
        # контроль на наличие ошибок.
    } elsif (Sname =~ /^add_(.*)/) {
        my $class_to_add = XML::ComicsML->element2class($1);
        my $object = $class_to_add->new;
        $self->appendChild($obJect);
        return $object;

    } elsif (Sname =~ /^remove_(.*)/) {
```

```
            my ($kid) = @_;
            $self->removeChild($kid);
            return $kid;
        }
}
# Заглушки.
sub elements {
    return ();
}
sub attributes {
    return ();
}
package XML::ComicsML::Comic;
use base qw(XML::ComicsML::Element);
sub elements {
    return qw(version title icon description url);
}
sub new {
    my $class = shift;
    return $class->SUPER::new('comic');
}
sub strips {
    # Возвратить список всех объектов, которые являются
    # потомками этого комикса.
    my Sself = shift;
    returnmap {XML::ComicsML->rebless($_)} $self->findnodes("./strip");
}
sub get_strip {
    # Задан ID, выборка полосы с этим атрибутом 'id'.
    my Sself = shift;
    my (Sid) = @_;
    unless (Sid) {
        warn "get_strip needs a strip id as an argument!";
        return;
    }
    my (@strips) = $self->findnodes("./strip[attribute::id='$id']");
    if (@strips > 1) {
        warn "Uh oh, there is more than one strip with an id of $id.\n";
    }
    return XML::ComicsML->rebless($strips[0]);
}
```

Гораздо больше классов элементов существует в реальной версии ComicsML, которая имеет дело с людьми, кадрами в комиксах, панелями кадров и тому подобное. Далее в этой главе объясняются упомянутые термины, которые будут применяться для решения реальной задачи.

XSLT: преобразование кода XML в HTML

Если вы когда-либо занимались web-крекингом при помощи Perl, то отчасти использовали возможности XML. Это связано с тем, что HTML не слишком далеко ушел от целей обеспечения формальной корректности, присущих XML, по край-

ней мере, теоретически. На самом деле, язык HTML часто используется в качестве комбинации разметки, знаков препинания, встроенных сценариев и многого другого из того, что делает создание web-страниц увлекательным занятием (наиболее популярные web-браузеры снисходительны к особенностям применяемого синтаксиса).

В настоящее время и, возможно, еще довольно долго универсальным языком Сети будет оставаться HTML. Хотя и существует возможность применения в web-страницах истинного языка XML, лежащего в основе XHTML[1], предложенного комитетом W3C, вероятнее всего придется преобразовать его в HTML-код в случае потребности в применении XML для Web.

Этот процесс можно рассматривать во многих ракурсах. Наиболее продуктивный из них заключается в разборе документа и получении результатов в виде CGI-сценариев. В этом примере считывается локальный файл MonkeyML, содержащий имена моих любимых обезьян, и выводится web-страница на стандартное устройство вывода (используется вездесущий CGI-модуль Линкольна Стейна (Lincoln Stein), придающий синтаксису некую «перчинку»):

```
#!/usr/bin/perl
use warnings;
use strict;
use CGI qw(:standard);
use XML::LibXML;
my $parser = XML::XPath;
my $doc = $parser->parse_file('monkeys.xml');
print header;
print start_html("My Pet Monkeys");
print h1("My Pet Monkeys");
print p("I have the following monkeys in my house:");
print "<ul>\n";
foreach my $name_node ($doc->documentElement->findnodes("//mm:name")){
     print "<li>" . $name_node->firstChild->getData ."</li>\n";
}
print end_html;
```

При реализации другого подхода применяется XSLT.

Средства XSLT используются с целью преобразования одного типа XML-кода в другой тип. Здесь XSLT играет весьма важную роль, поскольку работа происходит с XML, а Web часто требует применения HTML в качестве оболочки для важных фрагментов информации, извлеченных из XML-документов. В качестве примера можно рассматривать одно из XML-приложений, написанное на весьма высоком уровне. Это приложение называется AxKit (http://www.axkit.org), написано Мэттом Сержантом и основано на единой структуре сервера приложений. В этом

[1] Язык XHTML может выступать в двух ипостасях. Мы предпочитаем использовать менее требовательную «переходную» форму, трактующую «откровенные грехи» в качестве альтернативного метода (например, использование тега <font> вместо предпочтительного метода применения каскадных стилей).

случае можно настроить WWW-узел, который использует XML-код в виде файлов-источников. При использовании web-браузеров выводится HTML-код (а при использовании каких-либо других устройств информация выводится в любом, наиболее приемлемом формате).

Пример: Apache::DocBook

А теперь создадим небольшой модуль, который «на лету» преобразует файлы DocBook в HTML-файл. Хотя наша цель не столь претенциозна, как у AxKit, все же будем принимать команды от этих программ, основывая нашу программу на модуле Apache, mod_perl. С помощью данного модуля Perl-интерпретатор подключается к web-серверу Apache, и поэтому позволяет создать Perl-программу, которая выполняет все виды действий, применяемых по отношению к запросам к серверу.

Воспользуемся базовыми возможностями mod_perl, создав модуль Perl с подпрограммой-обработчиком, которая по стандартному имени осуществляет обратный вызов. Это имя передается объекту, представляющему Apache-запрос, а на базе этого объекта определяется, что будет видеть пользователь на экране.

СОВЕТ

Довольно часто, при выполнении программ на Perl и XML в среде Apache, пользователи испытывают некоторое разочарование. Причина этого кроется в самом Apache или, по крайней мере, в способе функционирования этого web-сервера (разумеется, если при компиляции не были даны отдельные указания на сей счет). Стандартный дистрибутив Apache включает С-библиотеки Expat, встроенные в бинарный код, если явно не указано другое. К сожалению, эти библиотеки часто конфликтуют с вызовами модуля XML::Parser библиотек Expat где-то в другом месте системы, приводя к трудно устранимым ошибкам (например, ошибки сегментации в Unix).

По имеющимся сведениям, сообщество разработчиков Apache рассматривает исключение этой возможности в будущих версиях, хотя в настоящее время она используется Perl-хакерами, применяющими Expat (как правило, посредством XML::Parser) с целью повторной компиляции Apache. Результатом этой операции будет отказ от использования Expat (параметру настройки EXPAT присваивается значение no).

Более дешевый прием заключается в использовании низкоуровневого модуля синтаксического разбора, который не применяет Expat, например, XML::LibXML, либо более современных модулей из семейства XML::SAX.

Начнем со «старта по типу модуля», затем перейдем к подпрограмме обратного вызова:

```
package Apache::DocBook;

use warnings;
use strict;

use Apache::Constants qw(:common);

use XML::LibXML;
use XML::LibXSLT;
our $xml_path; # Каталог документа источника.
our $base_path;# Каталог HTML-вывода.
our $xslt_file;# Путь к таблице стилей.
               # XSLT DocBook-в-HTML.
our $icon_dir; # Путь к значкам, использующимся на
               # индексных страницах.

sub handler {
    my $r = shift; # объект запроса Apache.
```

```
      # Получить информацию о конфигурации из
      # Apache-файла config
   $xml_path = $r->dir_config('doc_dir') or
die "doc_dir variable not set.\n";
   $base_path = $r->dir_config('html_dir') or die "html_dir variable not
set.\n";
   icon_dir = $r->dir_config('icon_dir') or die "icon_dir variable not
set.\n";
      unless (-d $xml_path) {
   $r->log_reason("Can't use an xml_path of $xml_path: $!", $r->filename);
   die;
   }
    my $filename = $r->filename;
      $filename =~ s/$base_path\/?//;
      # Добавить в путь информацию
      # (файл может в действительности не
      # существовать... ЕЩЕ)
   $filename .= $r->path_info;
   $xslt_file = $r->dir_config('xslt_file') or die "xslt_file Apache variable
not set.\n";

      # Будет вызвана подпрограмма, после чего произойдет
      # вывод на печать.

      # Это запрос индекса?
     if ( (-d "$xml_path/$filename") or ($filename =~ /index.html?$/) ) {
      # Почему бы нет! Мы выхватили индексную страницу.
      my ($dir) = $filename =~ /^(.*) (\/index.html?)?$/;
      # Полдела: добавить замыкающую косую черту к URI.
      if (not($2) and $r->uri !~ /\/$/) {
           $r->uri($r->uri '/');
           }
           make_index_page($r, $dir);
           return $r->status;
      } else {
      # Нет, это запрос какой-то другой страницы.
      make_doc_page($r, $filename);
      return $r->status;
      }
      return $r->status;
}
```

Эта подпрограмма выполняет фактическое преобразование в стиле XSLT, отображая имя исходного файла XML-источника и имя другого файла, в который следует записывать преобразованный HTML-вывод.

```
sub transform {
      my (Sfilename, $html_filename) = @_;
      # Убедитесь, что есть место для этого файла.
      maybe_mkdir($filename);

      my Sparser = XML::LibXML->new;
      my $xslt = XML::LibXSLT->new;

      # Поскольку библиотека libxslt выглядит
      # слегка поврежденной, нужно перейти в каталог
      # XSLT-файла, иначе включение этого файла невозможно. ;b
      use Cwd; # Так можно получить текущий рабочий каталог.
      my $original_dir = cwd;
      my $xslt_dir = $xslt_file;
```

```
        $xslt_dir =~ s/^(.*)\/.*$/$1/;
        chdir($xslt_dir) or die "Can't chdir to $xslt_dir: $!";
        my $source = $parser->parse_file("$xml_path/$filename");
        my $style_doc = $parser->parse_file($xslt_file);
        my $stylesheet = $xslt->parse_stylesheet($style_doc);
        my $results -$stylesheet->transform($source);
        open (HTML_OUT, ">$base_path/$html_filename");
        print HTML_OUT $stylesheet->output_string($results);
        close (HTML_OUT);
        # Вернуться к исходному каталогу.
        chdir($original_dir) or die "Can't chdir to $original_dir: $!";
}
```

Теперь рассмотрим пару подпрограмм, предназначенных для формирования индексных страниц. В отличие от страниц документа, которые являются результатом XSLT-преобразования, индексные страницы основаны на временном, большом массиве содержимого документа, которое было таблицей, наполненной информацией, захваченной из документа посредством XPath. Сначала проверяется соответствующий элемент метаданных, затем происходит обращение к другим битам информации (при отсутствии метаданных).

```
sub make_index_page {
            my ($r, $dir) = @_;
            # Если нет соответствующего каталога в XML-источнике,
            # запрос терпит неудачу.
        my $xml_dir = "$xml_path/$dir";
        unless (-r $xml_dir) {
            unless (-d $xml_dir) {
            # Увы, это не каталог.
                $r->status( NOT_FOUND );
                return;
            }
            # Это каталог, но его нельзя прочитать. Ничего.
            $r->status( FORBIDDEN ) ;
            return;
        }
        # Выбрать mtimes из этого каталога и index.html
        # в соответствующем каталоге html.
        my $index_f1le = "$base_path/$dir/index.html";
        my $xml_mtime = (stat($xml_dir))[9];
        my $html_mtime = (stat($index_file))[9];
        # Если индексная страница старше, чем XML-каталог, или
        # она просто не существует, сформировать новую страницу.
        if ((not($html_mtime)) or ($html_mtime <= $xml_mtime)) {
            generate_index($xml_dir, "$base_path/$dir", $r->uri);
            $r->filename($index_file);
            send_page($r, $index_file);
            return;
        } else {
            # Помещенная в кэш индексная страница подходит.
            # Пусть Apache обработает ее.
            $r->filename($1ndex_file);
            $r->path_info('');
            send_page($r, $index_file);
```

```
            return;
        }
}
sub generate_index {
    my ($xml_dir, $html_dir, $base_dir) = @_;
    # Удалить возможный завершающий / из base_dir.
    $base_dir =~ s|/$||;
    my $index_file = "$html_dir/index.html";
    my $local_dir;
    if ($html_dir =~ /^$base_path\/*(.*)/) {
        $local_dir = $1;
    }

    # Создать каталоги в случае необходимости.
    maybe_mkdir($local_dir);
    open (INDEX, ">$index_file") or die "Can't write to $index_file: $!";
    opendir(DIR, $xml_dir) or die "Couldn't open directory $xml_dir: $!";
    chdir($xml_dir) or die "Couldn't chdir to $xml_dir: $!";
    # Установить файлы, содержащие значки.
    my $doc_icon = "$icon_dir/generic.gif";
  my $dir_icon = "$icon_dir/folder.gif";
    # Создать визуализуемое имя $local_dir (возможно похожее).
    my $local_dir_label = $local_dir || 'document root';
    # Вывести на печать начало страницы.
    print INDEX <<END;
<html>
<head><title>Index of $local_dir_label</title></head>
<body>
<hl>Index of $local_dir_label</h1>
<table width="100%">
END
    # Вывести на печать по одной строке на файл в
    # этом каталоге
  while (my $file = readdir(DIR)) {
      # Игнорировать файл с точкой & каталоги &
      # служебные символы
      if (-f $file && $file !~ /^\./) {
        # Создать объекты анализатора.
        my Sparser = XML::LibXML->new;
        # Проверить формальную корректность,
        # отбросить, если не подходит:
        eval {$parser->parse_file($file);};
        if ($@) {
            warn "Blecch, not a well-formed XML file.";
            warn "Error was: $@";
            next;
        }
        my $doc = $parser->parse_file($file);
        my %info; # Будет хранить
                  # презентабельную информацию.
        # Определить тип корня.
        my $root = $doc->documentElement;
```

```
my $root_type = $root->getName;
# Теперь попробуем получить
# соответствующую информацию о вершине,
# $FOOinfo
my ($info) = $root->findnodes("${root_type}info");
if ($info) {
     # Элемент info. Присвоим ему значение с
     # помощью параметра %info.
     if (my ($abstract) = $info->findnodes('abstract')) {
          $info{abstract} = $abstract->string_value;
     } elsif ($root_type eq 'reference') {
          # Мы можем использовать первый refpurpose
          # вместо abstract.
     if ( ($abstract) = $root->findnodes('/reference/refentry/
          refnamediv/refpurpose')) {
          $info{abstract} = $abstract->string_value;
     }
}
if (my ($date) = $info->findnodes('date')) {
     $info{date} = $date->string_value;
     }
}
if (my ($title) = $root->findnodes('title')) {
     $info{title} = $title->string_value;
}
# Заполнить %info, нам не нужен XML для...
unless ($info{date}) {
     my $mtime = (stat($fHe)) [9];
     $info{date} = localtime($mtime);
}
$info{title} ||= $file;
# Достаточно info. Построим строку таблицы HTML.
     print INDEX "<tr>\n";
# Установим имя файла для связи с -- foo.html.
my $html_file = $file;
$html_file=~s/^(.*)\..*$/$1.html/;
print INDEX "<td>";
print INDEX "<img src=\"$doc_icon\">" if $doc_icon;
print INDEX "<a href=\"$base_dir/$html_file\">$info{title}</a></td> ";
 foreach (qw(abstract date)) {
 print INDEX "<td>$info{$_}</td> " if $info{$_};
 }
 print INDEX "\n</tr>\n";
 } elsif (-d $file) {
 # Просто выполним компоновку каталогов.
 # ...за исключением игнорируемого каталога.
 next if grep (/^$file$/, qw(RCS CVS .)) or ($file eq '..'
      and not $local_di r);
 print INDEX "<tr>\n<td>";
 print INDEX "<a href=\"$base_dir/$file\"><img
      src=\"$dir_icon\">" if $dir_icon;
 print INDEX "$file</a></td>\n</tr>\n";
 }
}
```

```
    # Закроем таблицу и завершим страницу.
    print INDEX <<END;
</table>
</body>
</html>
END
    close(INDEX) or die "Can't close $index_file: $!";
    closedir(DIR) or die "Can't close $xml_dir: $!";
}
```

Эти подпрограммы основаны на подпрограмме преобразования, выполняемого посредством формирования страниц. Обратите внимание на использование кэширования: при этом сравнивается время создания/модификации файла-источника DocBook и файла-получателя HTML, и последний перезаписывается только в том случае, если он старше, чем первый. (Конечно, если отсутствует HTML-код, всегда создается новая web-страница.)

```
sub make_doc_page {
    my ($r, $html_filename) = @_;
    # Формируем имя файла источника, заменяя
    # существующее расширение файла расширением .xml.
    my $xml_filename = $html_filename;
    $xml_filename =~ s/^(.*)(?:\..*)$/$1.xml/;

    # Если есть проблемы с чтением XML-файла источника, то это
    # связано с результирующим HTML.
    unless (-r "$xml_path/$xml_filename") {
        unless (-e "$xml_path/$xml_filename") {
        $r->status( NOT_FOUND );
        return;
    } else {
        # Существует, но нет прав чтения, не
        # обращать внимания.
        $r->status( FORBIDDEN );
        return;
        }
    }

    # Выбрать mtimes из этого файла и соответствующего
    # html-файла.
    my $xml_mtime = (stat("$xml_path/$xml_filename"))[9];
    my $html_mtime = (stat("$base_path/$html_filename"))[9];
    # Если html-файл старше, чем XML-файл, или если он просто
    # не существует, создать новый файл.

    if ((not($html_mtime)) or ($html_mtime <= $xml_mtime)) {
        transform($xml_filename, $html_filename);
        $r->filename("$base_path/$html_filename");
        $r->status( DECLINED );
        return;
    } else {
        # Он помещается в кэш. Пусть Apache использует
        # существующий файл.
        $r->status( DECLINED );
    }
}
```

```
sub send_page {
    my ($r, $html_filename) = @_;
    # Есть проблема: если мы создаем файл, мы не можем
    # записывать в него и сказать 'ОТМЕНИТЬ', поскольку
    # по умолчанию сервер обрабатывает подобные обращения,
    # отображая сообщения "файл-не-найден". Пока я не
    # нашел лучшего решения, я просто отображаю содержимое
    # файла и выполняю DECLINE только для известных
    # кэш-обращений.
    $r->status( OK );
    $r->send_http_header('text/html');

    open(HTML, "$html_filename")or
     die "Couldn't read $base_path/$html_filename: $ !";
     while (<HTML>) {
     $r->print($_);
     }
     close(HTML) or die "Couldn't close $html_filename: $!";
     return;
}
```

Наконец, существует подпрограмма-утилита, помогающая в трудоемкой задаче: копирование структуры подкаталогов в кэш-каталог, который отображает каталог-источник XML:

```
sub maybe_mkdir {
     # Задан путь, убедитесь, что приводящие к нему
     # каталоги существуют, создайте каталог, если он
     # не существует.
     my ($filename) = @_;
     my @path_parts = split(/\//, $filename);
     # Если последний объект - имя файла, отбросьте его.
     pop(@path_parts) if -f $filename;
     my $traversed_path = $base_path;
     foreach (@path_parts) {
          $traversed_path .= "/$_";
          unless (-d $traversed_path) {
           mkdir ($traversed_path) or die "Can't mkdir $traversed_path: $!";
               }
     }
    return 1;
}
```

Индекс комиксов

Язык XSLT имеет ограниченные возможности, в то время как потенциал совместной обработки средствами Perl, XML и Web поистине безграничен, как и объекты, которые могут обрабатываться с их помощью. Иногда недостаточно просто поставить клиентам разобранный XML-код, а необходимо написать Perl-модуль, который «выжимает» из XML-документов интересную информацию и строит нечто в стиле Web, не связанное с требуемым результатом. Нечто похожее было проделано в предыдущем примере, когда применялся исходный язык XSLT при преобразовании документов DocBook с одновременным созданием индексной страницы.

Поскольку мы столкнулись с проблемами применения таких XML-методов, как RSS и ComicsML в этой и предыдущей главе, то напишем небольшую программу, функционирующую в web-стиле. С целью иллюстрации применяемой идеи, созда-

дим простую CGI-программу, которая формирует индекс любимых интерактивных комиксов пользователя (в воображаемом мире у всех есть соответствующие документы ComicsML):

```
#!/usr/bin/perl

# Очень простой обработчик ComicsML; задан список
# URL-ссылок, указывающих на ComicsML-документы, выбрать
# их, выровнять их сценарии в один список, а
# затем сформировать листинг web-страницы, связывая и,
# возможно, отображая на экране сами сценарии, упорядочив,
# начиная от самого последнего сценария.

use warnings;
use strict;

use XML::ComicsML;     # ...так что мы можем создать объекты ComicsML
use CGI qw(:standard);
use LWP;
use Date::Manip; # Поскольку мы слишком ленивые, чтобы самим сравнивать даты

# Предположим, что URL моих любимых ComicsML-документов
# в Интернете находятся в незашифрованном текстовом файле
# на диске, с одним URL на строку.
# (Что, никакого XML? Позор...).

my $url_file = $ARGV[0] or die "Usage: $0 url-file\n";

my @urls;                 # список URL на ComicsML-документы
open (URLS, $url_file) or die "Can't read $url_file: $!\n";
while (<URLS>) { chomp; push @urls, $_; }
close (URLS) or die "Can't close $url_file: $!\n";

# Создать пользовательский LWP-агент.
my $ua = LWP::UserAgent->new;
my Sparser = XML::ComicsML->new;

my @strips; # Здесь содержатся объекты, представляющие комиксы.

foreach my $url (@urls) {
     my $request = HTTP::Request->new(GET=>$url);
     my $result = $ua->request($request);
     my $comic;             # Сохраним комикс, мы еще вернемся
     if ($result->is_success) {
     # Проверим, понравится ли это ComicsML-анализатору.
     unless ($comic = $parser->parse_string($result->content)) {
          # Увы, это неподходящий XML-документ.
          warn "The document at $url is not good XML!\n";
          next;
          }
     } else {
     warn "Error at $url: " . $result->status_line . "\n";
     next;
     }
     # Теперь очистим все сценарии от комиксов, поместив
     # в каждый несколько хэш-ссылок вместе с информацией
     # о самом комиксе.
     foreach my $strip ($comic->strips) {
          push (@strips, {strip=>$strip, comic_title=>$comic->title,
        comic_url=>$comic->url});
     }
}
```

```
# Упорядочим список сценариев по дате. (Мы используем
# здесь экспортируемую функцию UnixDate Date::Manip,
# преобразуя даты григорианского календаря
# в даты, принятые в Unix).
my @sorted = sort {UnixDate($$a{strip}->date, "%s") <=>
UnixDate($$b{strip}->date, "%s")} @strips;

# Теперь мы построили web-страницу!

print header;
print start_html("Latest comix");
print h1("Links to new comics...");

# Пройдемся по упорядоченному списку в обратном порядке.
foreach my $str1p_info (reverse(@sorted)) {
    my ($title, $url, $svg);
    my $strip = $$strip_info{strip};
    $title = join (" - ", $strip->title, $strip->date);
    # Гиперссылка заголовка на URL-ссылку, если
    # таковая подготовлена.
    if ($url = $strip->url) {
        $title = "<a href='$url'>$title</a>";
    }

    # Аналогично поступаем с заголовками и
    # URL-ссылками комиксов.
    my $comic_title = $$strip_info{comic_title};
    if ($$strip_info{comic_url}) {
        $comic_title = "<a href='$$strip_info{comic_url}'>$comic_title</a>";
    }
    # Вывести на печать заголовки.
    print p("<b>$comic_title</b>: $title");
    print "<hr />";
    }
print end_html;
```

Учитывая сложности, с которыми мы столкнулись при использовании модуля Apache: : DocBook, эта программа может показаться слишком простой; она не выполняет кэширование, не включает регуляторы количества обрабатываемых полос, а ее «познания» в области структуры web-страниц поверхностны и довольно приблизительны. Хотя она великолепно работает и демонстрирует свои преимущества при использовании модулей-помощников типа XML: : ComicsML.

На этом завершается краткий обзор возможностей совместного применения Perl и XML. Как отмечалось в начале книги, взаимосвязь между этими двумя технологиями еще мало изучена, а возможный потенциал до конца еще не осознан. Во время работы над этой книгой появились новые анализаторы вида XML: : LibXML и новая концепция PerlSAX2. Мы надеемся, что предоставили достаточно информации и содействовали в том, чтобы вы стали участником этих событий, поскольку в ближайшие годы намечается громадный прогресс в этой области.

<aloha>Удачи во всех начинаниях!</aloha>

Алфавитный указатель

Иностранные термины

А

Б

Д

И

К

Л

М

Н

О

П

Эрик Рэй, Джейсон Макинтош
Perl & XML. Библиотека программиста
Перевели с английского И. Киричок, Н. Перевозчикова, А. Синева

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Редактор	*А. Сергеев*
Литературные редакторы	*Т. Криницкая, О. Янковая*
Художник	*Н. Биржаков*
Корректор	*Н. Ребенок*
Технический редактор	*Н. Ребенок*

Лицензия ИД № 05784 от 07.09.01.

Подписано в печать 22.11.02. Формат 70x100/16. Усл. п. л. 16,77.
Тираж 4000 экз. Заказ № 1816.
ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в.
Налоговая льгота - общероссийский классификатор продукции ОК 005-93, том 2; 953005 - литература учебная.
Отпечатано с диапозитивов в ФГУП «Печатный двор» им. А. М. Горького Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.